THE UNIVERSITY *of* EDINBURGH

Energy Efficient Cache Architectures for Single, Multi and Many Core Processors

Karthik Thucanakkenpalayam Sundararajan

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2013

Abstract

With each technology generation we get more transistors per chip. Whilst processor frequencies have increased over the past few decades, memory speeds have not kept pace. Therefore, more and more transistors are devoted to on-chip caches to reduce latency to data and help achieve high performance.

On-chip caches consume a significant fraction of the processor energy budget but need to deliver high performance. Therefore cache resources should be optimized to meet the requirements of the running applications. Fixed configuration caches are designed to deliver low average memory access times across a wide range of potential applications. However, this can lead to excessive energy consumption for applications that do not require the full capacity or associativity of the cache at all times. Furthermore, in systems where the clock period is constrained by the access times of level-1 caches, the clock frequency for all applications is effectively limited by the cache requirements of the most demanding phase within the most demanding application. This motivates the need for dynamic adaptation of cache configurations in order to optimize performance while minimizing energy consumption, on a per-application basis.

First, this thesis proposes an energy-efficient cache architecture for a single core system, along with a run-time support framework for dynamic adaptation of cache size and associativity through the use of machine learning. The machine learning model, which is trained offline, profiles the application's cache usage and then reconfigures the cache according to the program's requirement. The proposed cache architecture has, on average, 18% better energy-delay product than the prior state-of-the-art cache architectures proposed in the literature.

Next, this thesis proposes cooperative partitioning, an energy-efficient cache partitioning scheme for multi-core systems that share the Last Level Cache (LLC), with a core to LLC cache way ratio of 1:4. The proposed cache partitioning scheme uses small auxiliary tags to capture each core's cache requirements, and partitions the LLC according to the individual cores cache requirement. The proposed partitioning uses a way-aligned scheme that helps in the reduction of both dynamic and static energy. This scheme, on an average offers 70% and 30% reduction in dynamic and static energy respectively, while maintaining high performance on par with state-of-the-art cache partitioning schemes.

Finally, when Last Level Cache (LLC) ways are equal to or less than the number of cores present in many-core systems, cooperative partitioning cannot be used for partitioning the LLC. This thesis proposes a region aware cache partitioning scheme

as an energy-efficient approach for many core systems that share the LLC, with a core to LLC way ratio of 1:2 and 1:1. The proposed partitioning, on an average offers 68% and 33% reduction in dynamic and static energy respectively, while again maintaining high performance on par with state-of-the-art LLC cache management techniques.

Dedicated to my parents and my advisor

Acknowledgements

First and foremost, I wish to thank my advisor, Professor Nigel Topham for his wonderful guidance and support, throughout my PhD studies. His Microprocessors and Microcontrollers course that he taught during my Master's degree is responsible for much of what I know in computer architecture. He gave me the freedom to do the research in my chosen area, caches. Whenever I encountered a problem, he always stood by me and boosted my morale, as a result of which, I felt more encouraged and motivated to achieve my goals.

Secondly, I would like to thank Timothy M. Jones for providing valuable suggestions and feedback during my research. His interest in caches along with the technical expertise improved my knowledge on caches. The discussions that I have had with him, would always give me a new dimension to approach the issue in question. The most important lesson I have learnt from him is how to present the ideas with lucidity and coherence in the research article.

I would like to thank Vasileios, a very nice person and a good friend of mine. I used to have lot of late night and weekend discussions with him. Not only did he listen to my ideas with patience, but he would also offer valuable feedback. We shared a common interest for collecting radio controlled helicopters, which is our main stress-buster.

I was lucky to have Luis Fabricio Wanderley Goes as my friend. I would like to specially thank him and his wife, Raquel for inviting us to the christmas dinner and serving their traditional pão de queijo along with rice and beans.

I was gifted to have the New Texans Nikolas, Andrew, Bharghava, George, Konstantina, Murali, Kiran, Luna and Victor as my office mates. I can describe them in one word. They are just amazing. They acted as a strong pillar of support for my personal and research life.

I would like to mention that Mulari and his wife, Meenakshi, would take various short trips in and around Scotland. I would always cherish the memories of those trips.

I would like to thank Bjorn Franke, Mike O'Boyle, Murray Cole and Vijay Nagarajan for the support they gave me during my PhD. Their support is invaluable.

Last but not the least, I would like to thank my parents and my wife Pavithra, without whom, I wouldn't be successful in my endeavours.

Declaration

I declare that this thesis was composed by myself, that the work obtained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

- Karthik T. Sundararajan, Timothy M. Jones and Nigel P. Topham. *A Reconfigurable Cache Architecture for Energy Efficiency*. (Poster paper). In Proceedings of the 8^{th} ACM International Conference on Computing Frontiers (CF'11), Ischia, Italy, May 3-5, 2011.

- Karthik T. Sundararajan, Timothy M. Jones and Nigel P. Topham. *Smart Cache: A Self Adaptive Cache Architecture for Energy Efficiency*. In Proceedings of the 11^{th} IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'11), Samos, Greece, July 19-22, 2011.

- Karthik T. Sundararajan, Vasileios Porpodas, Timothy M. Jones, Nigel P. Topham and Bjorn Franke. *Cooperative Partitioning: Energy-Efficient Cache Partitioning for High-Performance CMP's*. In Proceedings of the 18^{th} IEEE International Symposium on High Performance Computer Architecture (HPCA'12), New Orleans, Louisiana, February 25-29, 2012.

- Karthik T. Sundararajan, Timothy M. Jones and Nigel P. Topham. *Energy-Efficient Cache Partitioning For Future CMPs*. (Poster paper). In Proceedings of the 21^{st} ACM International Conference on Parallel Architectures and Compilation Techniques (PACT'12), Minneapolis, Minnesota, September 19-23, 2012.

- Karthik T. Sundararajan, Timothy M. Jones and Nigel P. Topham. *The Smart Cache: An Energy-Efficient Cache Architecture Through Dynamic Cache Adaptation*. In the International Journal of Parallel Programming(IJPP'13): Volume 41, Issue 2 (April 2013), Page 305-330.

(*Karthik Thucanakkenpalayam Sundararajan*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computers have evolved tremendously over years, from the difference engine to transistors. This evolution has seen an increase in computing performance with a reduction in size of the computing unit. With the introduction of the first microprocessor from Intel in 1971 to recent embedded and high performance systems, advancement in semiconductor technologies has seen the transistor count per silicon area double every eighteen months in accordance with Moore's law [96], as stated by Gordon E. Moore.

This increase in transistor count per silicon area has led to the addition of more circuitry to improve performance. Such circuitry includes the addition of on-chip caches, branch predictors, deep pipelines, out-of-order, superscalar engines and speculative execution. With the addition of these, the single-core performance has improved considerably and has now reached a point where a small amount of performance improvement requires a significant number of transistors. The failure of Dennard scaling (Dennard et al. [28]) has meant that complex designs require too much power. So performance now comes from multiple, simpler cores. Moreover the design has also become more complex with current-swings and signal noise leading to the end of the single core system.

The most effective way of utilizing the available transistors is to add more processing cores on-chip, beginning a new-era of chip multiprocessors (CMPs). CMP achieves more performance over a single core system by exploiting thread level parallelism (TLP). CMPs mitigate the design complexity and thermal issues present in a single core by uniformly spreading the computational resources across the chip as parts of the different cores in the CMP. Some of the recent examples of CMPs include, four-core Intel i7, six-core AMD Phenom X2, eight-core UltraSPARC T2, eight-core AMD Bulldozer etc,.

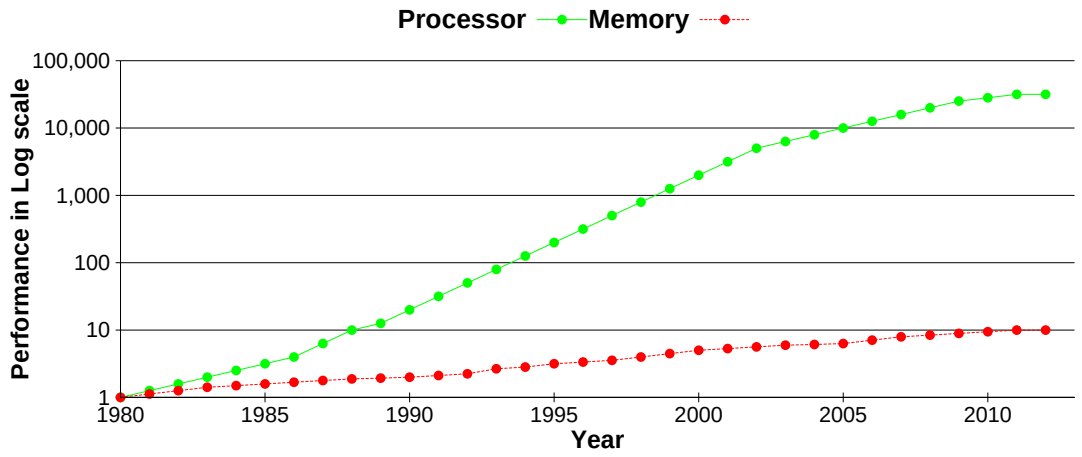


Figure 1.1: Example showing the gap in performance between the processor and memory [49].

1.1 The Problem

Over recent years, processor speeds have increased at a faster rate than DRAM speeds [49]. This trend is shown in Figure 1.1. Current generation processors have main memory access latency of more than 300 processor cycles and projections show that this will increase in near future [133]. The memory acts as a barrier/wall to achieve more performance. This is often referred as the *Memory Wall [135]*.

To bridge the gap between the processor and main memory, caches are used to keep frequently-used data needed by the cores. Performance improvement is achieved by serving the request directly from caches, which are faster than DRAM.

Due to varying cache requirement of applications, the working sets of some applications fit in a small cache, which is more energy efficient, and some require a large cache to fit the working set. Hence designers use a hierarchy of caches to strike a balance between small and large cache size. The caches closer to the processor are smaller and faster since they are optimised for speed. Caches further away are slower and optimised for energy or area.

First level caches have an impact on the processor access time, as they will be accessed first and also need to serve the incoming request at a faster rate compared to Last Level Cache (LLC). Hence they need to be designed effectively than the LLC. Comparatively, LLCs do not have the strict access time constraints of first level caches. A miss in the LLC will access the main memory, which will take hundreds of processor cycles. Hence they need to be effectively managed.

As technology advances in CMOS, the power dissipation of modern micropro-

cessors is a primary design constraint across all processing domains, from embedded devices to high performance chips. Shrinking feature sizes and increasing numbers of transistors packed onto a single die only exacerbate this issue. Schemes are urgently required to tackle power dissipation, yet still deliver high performance from the system. Cache memories occupy the majority of chip area and are a key target for energy reduction. For instance, 60% of the StrongARMs area is devoted to caches [88].

In a single core system, fixed conguration caches must be designed to deliver low average memory access times across a wide range of potential applications. However, this can lead to excessive energy consumption for applications that do not require the full capacity or associativity of the cache at all times. Customizing the cache parameters like cache size, line size and associativity according to an application, yields better utilization of the cache. Prior cache architectures in the literature were good at offering static reconfiguration, that is, finding the best cache parameters for an application through offline profiling and then running the application with those predetermined configurations yielding better power and performance when compared to running them with a fixed baseline configuration. However, reconfiguring the cache according to the phase of an application requires new cache architectures to support effective runtime reconfiguration without completely flushing the cache between phases.

In a multi-core system, with a core to LLC way ratio of 1:4, each core has a private first level cache and they all share the LLC. Intelligently partitioning the last-level cache within a chip multiprocessor can bring significant performance improvements. Resources are given to the applications that can benefit most from them, restricting each core to a number of logical cache ways. However, although overall performance is increased, existing schemes fail to consider energy saving when making their partitioning decisions. When each core executes a different application, commonly known as a multi-programmed scenario or when it executes different regions of the same program, commonly known as a multi-threaded scenario, their memory requirement varies. Hence the LLC needs to be effectively shared among the cores. Existing cache partitioning schemes in the literature have partitioned the LLC for improved performance. As the technology advances in CMOS, static energy dissipation will be dominant and require more attention than dynamic energy.

In a many-core system, as the number of processing cores increases, the core-to-way ratio in the last level cache (LLC) becomes 1:2 or 1:1, presenting problems to existing cache partitioning techniques which require more ways than cores. Further, effective energy management of the LLC becomes increasingly important due to its

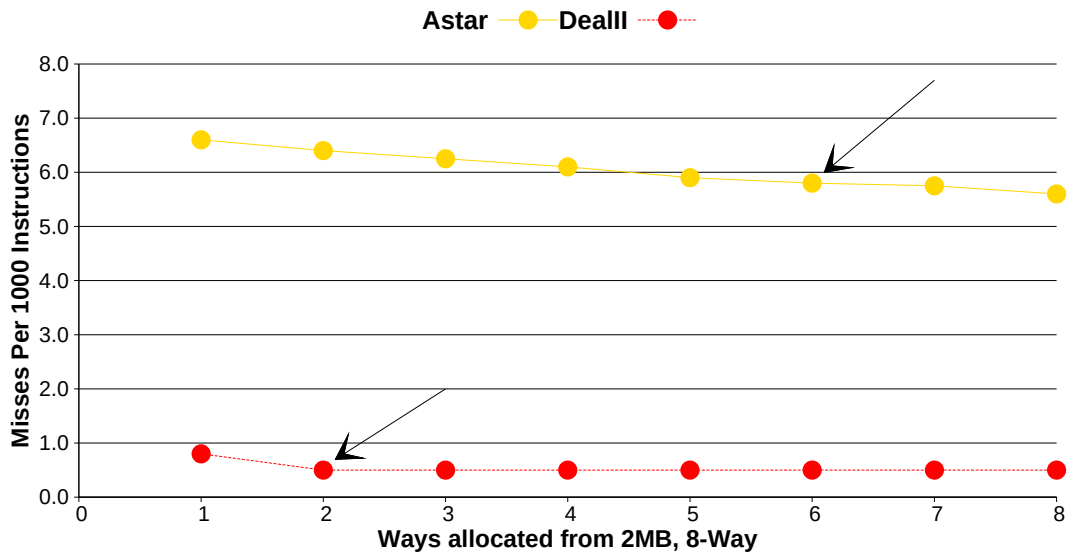


Figure 1.2: Example showing the heavy cache requirement of two applications that share a LLC.

size. Most previous work on cache energy savings are based on single-core designs and are either focused on turning off parts of the cache memory to save static energy or on predicting the way that will be accessed, thereby saving dynamic energy. However, these techniques are mostly inapplicable to the LLC.

1.2 Motivation

The previous section described the need for an energy efficient cache management schemes. This section motivates the need by presenting two examples. First, it presents an example of a fully utilized cache. Second, it presents an example of an under utilized cache.

1.2.1 LLC - fully utilized

Figure 1.2 shows two applications, *astar* and *dealii* from SPECCPU2006 [122] benchmark that runs on a two-core system. The Y-axis represents Misses Per 1000 Instructions (MPKI) and X-axis represents the number of ways allocated from a 2MB, 8-way LLC cache. In a two-core system, one core is kept idle, an application, for example, *astar* is executed on the remaining core. The MPKI value for *astar* application that runs with the cache ways from one to eight in a 2MB 8-way LLC cache are plotted as shown in Figure 1.2. The same process is repeated for *dealii* application and the MPKI

values are plotted.

When *astar* and *dealii* applications run in a two core system that share a LLC, both applications compete for the LLC cache space by one application trying to evict others' data, leading to a performance degradation. This shows a need for the cache partitioning scheme that identifies each application's cache requirement and partitions the cache accordingly.

The cache partitioning should be able to identify each applications' cache requirement and partition the cache according to its requirement. Assuming, the partition decision is made and is shown in arrows in Figure 1.2. Accordingly, the *astar* application gets six ways and *dealii* gets two ways. When more cache ways are given to *dealii* application, the performance improvement achieved over the allocation of extra ways is less than 2%. Therefore, providing more cache ways beyond this results in diminishing improvement in performance.

This thesis proposes an energy efficient cache partitioning scheme that activates only the required cache ways that has the application's data. This leads to the reduction in dynamic power. Since all the cache ways are utilized, the static power is not reduced.

1.2.2 LLC - under utilized

Figure 1.3 shows two applications, *xalan* and *dealii* that run in a two core system. These applications do not benefit from a big cache. Assuming, the partition decision is made and is shown in arrows in Figure 1.2. Accordingly, *xalan* application gets three ways and *dealii* gets two ways. Providing more cache ways beyond this, results in diminishing improvement in performance.

This thesis proposes an energy efficient cache partitioning scheme that upon receiving the cache access request, activates only the required cache ways that has the requesting application's data. This reduces the dynamic power by activating only the required cache ways. The remaining three cache ways will be turned-off to reduce the static power. Thus both dynamic and static power reduction is achieved by using an energy efficient cache partitioning scheme.

1.3 Contributions

This dissertation makes the following contributions:

- This dissertation presents a new reconfigurable cache architecture namely, Smart

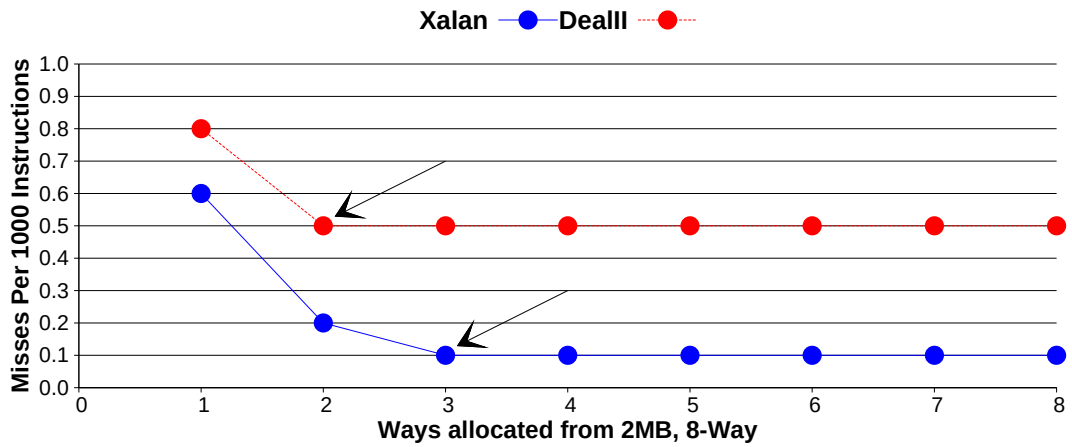


Figure 1.3: Example showing the minimal cache requirement of two applications that share a LLC.

cache for the single core system to support effective runtime reconfiguration, by avoiding complete flushing of the data between program phases. The Smart cache organizes cache way at set boundaries, which avoids flushing of data back to memory when increasing the associativity but keeping the cache size fixed. The energy-delay of the proposed Smart cache is on average 18% better than state-of-the-art cache reconfiguration architectures. A decision tree based machine learning model that predicts the cache configurations for the current program phase based on its observation from the previous phase is used to drive the proposed cache architecture. Smart cache offers an average reduction in energy-delay product of 17% in the data cache (just 1% away from an oracle result) and 34% in the level-2 cache (just 5% away from an oracle), with an overall performance degradation of less than 2% compared with a baseline statically-configured cache. The Smart cache is most suitable for embedded systems that have a single processing core and are power efficient.

- This dissertation presents an energy efficient cache partitioning technique, namely cooperative partitioning for a multi-core CMP that has a core-to-LLC way ratio of 1:4. The assumption of this core-to-LLC way ratio holds for desktop systems with two or four cores, sharing an 8-way or a 16-way LLC respectively. Commercial examples that have this core-to-LLC way ratio include Intel's two-core Core2Duo and Intel's four-core i7. Cooperative partitioning uses shadow tags (32 sampled sets from LLC) to monitor the LLC requirement of each application running on an individual core and LLC partition decisions are taken accordingly.

Results shows that cooperative partitioning offers an average dynamic and static energy saving of 35% and 25% respectively, compared to a fixed partitioning scheme. In addition, Cooperative Partitioning maintains high performance while transferring ways five times faster than an existing state-of-the-art technique.

- Finally, this dissertation presents the Region Aware Cache Partitioning (RECAP) technique for many-core CMP that has a core-to-LLC way ratio of 1:2 or 1:1. As there are equal or less number of cache ways available than the number of cores in CMP, the state-of-the-art cache partitioning techniques cannot be applied. Like the state-of-the-art partitioning schemes, RECAP monitors the LLC requirement of each application running on an individual core. However, it significantly differs from the rest by collectively grouping the application based on the cache requirement and left justifies the group. Results for a 8-core CMP running multiprogrammed and multi-threaded workloads, show that RECAP achieves 68% dynamic and 33% static energy savings and 77% dynamic energy and 60% static energy respectively from the shared LLC with no performance loss.

1.4 Structure

This thesis is organized into six chapters and are as follows

Chapter 2 presents background information. It presents the necessary background on caches, in terms of fundamental concepts, cache organization in single, multi and many core processors.

Chapter 3 presents the related work. It discusses the research on cache management for single, multi and many core processors.

Chapter 4 presents SMART cache, a new cache architecture for single core system. First, it discusses the related work in reconfigurable caches present in single core systems. Second, it presents the new architecture and evaluates its significance compared to the existing state-of-the-art cache architectures. Finally, it presents a decision tree based machine learning model to drive the proposed cache architecture. This chapter is based on the work published in [128, 126].

Chapter 5 presents cooperative partitioning, a cache partitioning scheme for multi-cores that share the LLC. First it presents a new cache partitioning framework. Second, it presents the prior work on cache partitioning. Third, it describes the simulation

environment and evaluation methodology. Finally it discusses related work in cache partitioning for multi-core systems. This chapter is based on the work published in [129].

Chapter 6 presents RECAP, a cache partitioning framework for future many core CMPs. First it presents the proposed cache partitioning framework. Second, it evaluates the proposed scheme to the existing state-of-the-art cache partitioning schemes. Third, it presents the evaluation methodology and result analysis. Finally, it discusses the related work in cache management for many-core CMPs. This chapter is based on the work published in [127].

Chapter 7 presents the summary and directions for future work.

1.5 Summary

This chapter has provided an overview of the problem associated with shared caches present in single, multi and many core processors. It has provided an outline of the thesis, specifically dedicating three chapters to target the problems associated with shared caches in single, multi and many core processors. The next chapter provides the background information on caches and their organization in single, multi and many core processors.

Chapter 2

Background On Caches And Machine Learning

This chapter provides a basic introduction to caches and describes various cache parameters that impact processor performance. This chapter also provides an introduction to machine learning models such as decision trees and linear regression models that are used in the thesis.

2.1 Terminology

The major limitation to processor performance has been access to main memory, which is two orders of magnitude slower than the processor. The term *cache* refers to a buffer that stores a small amount of recently-used data. It is usually placed between the processor and main memory, such that it stores/caches the frequently accessed data of a program. Caches exploit temporal locality by caching recently-used data, and spatial locality by caching data that is physically close to other used data. Therefore, it tries to mask the difference in speed between the processor and memory.

Due to cost and area constraints, caches cannot be large enough to hold the entire working set of most programs, therefore a design trade-off is made with respect to the available area and expected processor performance. Cache parameters like cache size, line size and associativity play a major role in determining the performance of the processor. Cache size refers to the available cache space in bytes. Line size refers to the cache line width in bytes. This is circled and shown in Figure 2.1. Cache associativity refers to the number of available cache ways in the cache.

Processor performance depends on both, cache hit and cache miss. A cache hit

occurs when a memory request is satisfied from the cache. A cache miss is classified as either, a capacity miss, a conflict miss or a compulsory miss [53]. Capacity misses is due to the working set of the program does not fit in the cache. These misses are reduced by increasing the cache size. Conflict misses occurring due to the working sets that compete for the same cache line and are reduced by increasing the cache associativity. Compulsory misses occur due to the cache being empty at the start of the program execution. It cannot be avoided, as the cache will be empty at the start.

Applications that have a small working set require a single level cache, whereas some other applications require a multi level caches to fit their working set. Hierarchies of caches have a fast access time and also consume less dynamic power compared to a single large cache. The processor designers use multi level caches to filter out the accesses to memory by serving the incoming request from the processor through the hierarchy of caches.

In a system with multi level caches, some of them are private, which means that only a particular core can access the cache and some of them are shared among cores, which means that all cores can access the shared cache. This is shown in Figure 2.5

Figure 2.1 shows a layout of a cache. It assumes a virtually indexed, physically tagged cache, which means that the virtual address is used for indexing in to the sets and physical address is used for tag comparison. The tag part stores the address and the data part stores the data. The incoming address from the processor, virtual address in this case, is split into three fields; tag, index and offset. There are two parts; tag part and data part.

First, the tag part is accessed. Each cache line stores the physical address from the tag field. The index field is used to select the sets. After selecting the set, the physical address present in the set is compared against the incoming tag field of the address, upon a hit, the corresponding way of the set present in data part is selected. Now, the offset field of the address is used to select within the selected set present in data part. A set in a Way forms a cache line/cache block as circled in Figure 2.1.

The cache access time and power consumption for a set-associative cache, will be more than that of a direct-mapped cache (a cache with just one way). This is due to the activation and searching of all the ways for every cache access. Caches are banked to reduce the dynamic power consumed in activating all the sets present in the entire cache way. In a direct-mapped or set-associative cache, instead of activating all the sets present in the entire way, only a subset of sets are enabled. Upper order index bits select the bank, in Figure 2.1, it is either bank 0 or bank 1 and lower order index bits

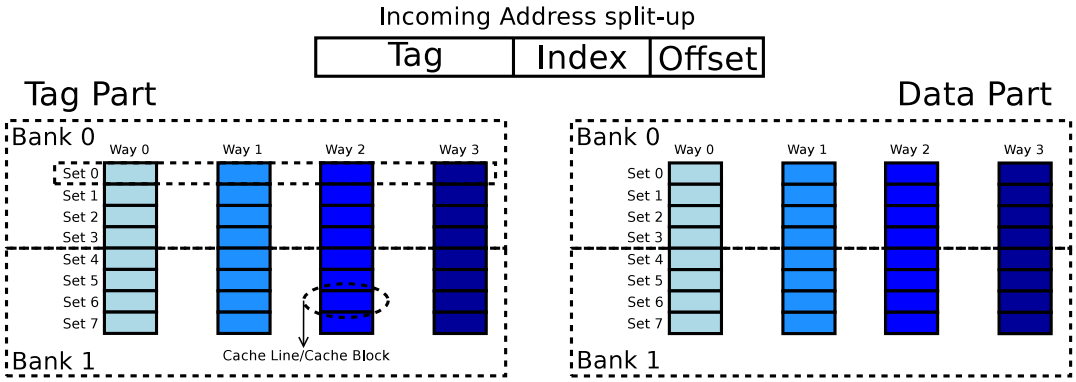


Figure 2.1: Diagram showing a sample layout of a Cache along with tag and data part.

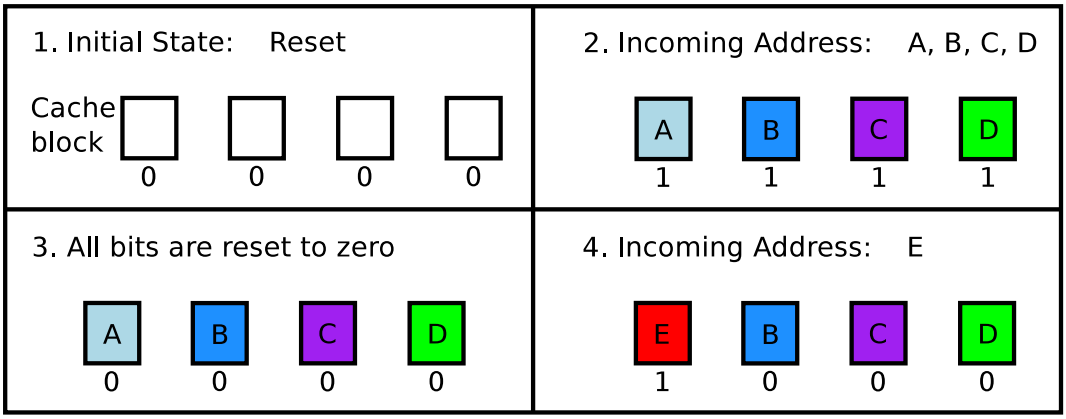


Figure 2.2: Example showing the working of PLRU replacement policy.

selects the set within the bank.

2.2 Replacement Policy

Identifying the right cache block for replacement improves processor performance. Least Recently Used (LRU) policy is the most widely used replacement policy. As the name suggest, it keeps track of the recently accessed blocks and identifies the last accessed block for replacement. The overhead for this scheme is the hardware required to track the recently accessed block for every set. Several other schemes have been proposed to approximate LRU's functionality with a reduced hardware, such as Pseudo LRU (PLRUm) [7].

PLRUm uses one bit for every cache block. Initially the bits are reset. When a cache block is accessed, the bit is set. When all bits in a set are set, all of them are reset.

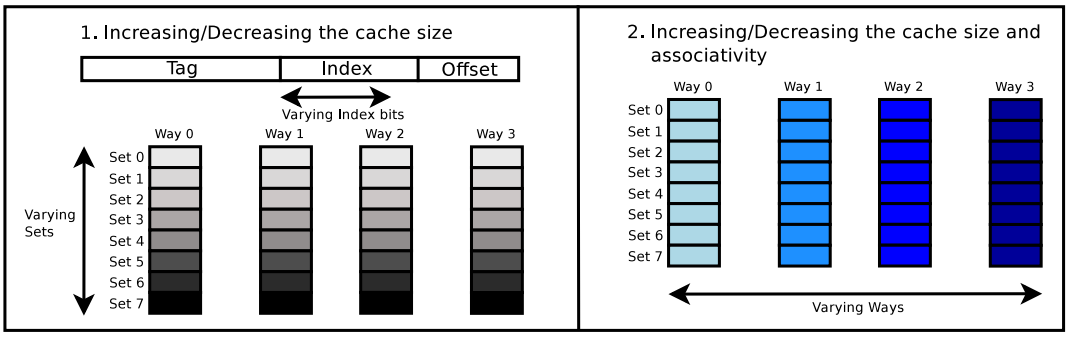


Figure 2.3: Example showing a cache reconfiguration process.

2.3 Reconfiguration

Figure 2.3 shows different types of cache reconfigurations. This example assumes a cache with eight sets and a four-way set-associativity cache.

First, the cache size is altered by varying the number of sets. This is achieved by varying the index bits present in the index field according to the required cache size. Usually, all the index bits are required to select the desired set. However, leaving out the upper order bit (most significant bit) of the index field, brings the cache down to half of its original size. Further, leaving out the next upper order bit of the index field, reduces the cache to a quarter of its original size. Varying the index field results in a change in the hash mapping function. For example, the original index field as show in Figure 2.3 selects a single cache set from the available cache sets 0-7, omitting the most significant bit from the index field, selects a set from set 0-3. A change in the mapping function is observed. Therefore, the data has to be flushed before reconfiguration. This can be an overhead.

Second, the cache size and associativity are altered by varying the number of cache ways. This does not require altering the index bits. Having a way enable/disable signal is sufficient to increase/decrease the cache size and associativity. Decreasing the number of cache ways not only decreases the associativity but also decreases the cache size. The data has to be flushed before reconfiguration. Since the hash mapping function is not changed, it requires a gradual flushing of data.

Both these schemes have their merits and de-merits. Chapter 4 presents a hybrid scheme that incorporates both schemes to offer maximum flexibility in cache reconfiguration and also discusses the potential overheads associated with combining these schemes. Combining both schemes offers a fine grain reconfiguration, compared to their individual coarse-grain reconfiguration.

2.4 Partitioning

Cache partitioning refers to partitioning the cache among the competing cores. Figure 2.5 shows examples of an LLC with and without cache partitioning schemes.

Figure 2.5.A shows the data layout of an unpartitioned cache. The data of application running on Core 0 and Core 1 have the freedom to occupy the available cache ways. The potential problem associated with an unpartitioned cache is inter-core interference. The de-facto replacement policy is a LRU policy, which allocates the cache block depending on the access frequency. When an application with a large working set tries to fit in a smaller LLC cache, it takes the entire cache space by evicting other application's data present in the LLC. This leads to a thrashing behaviour of one application evicting the data of other application. Therefore, a thrashing application running on Core 1, accessing the LLC more frequently, occupies a majority of the LLC cache space, leading to a poor allocation of cache space to application running on Core 0.

Figure 2.5.B shows the data layout of a partitioned cache. The data corresponding to application running on Core 0 and Core 1 can be placed only in the corresponding ways that have Core 0 and Core 1 data. For example, application running on Core 1 cannot have the data placed in way 0. Thus, cache partitioning potentially avoids the inter-core interference by limiting the placement of each individual core's data to a selected number of cache ways.

Cache partitioning for multi and many core processor is discussed in further detail in Chapters 5 and 6.

2.5 Machine Learning

Machine learning is a technique that allows machines/computers to infer knowledge. Depending on how they infer the knowledge, they are classified as supervised or unsupervised learning.

In supervised learning, the outputs are already known for the given inputs. A statistical analysis of the inputs and extracting information, known as feature extraction is carried out. These features are then used to predict the outputs from new inputs. Supervised learning techniques, such as linear regression and decision trees are discussed in next sections.

In unsupervised learning, there is no predetermined input or output available, unlike supervised learning. Based on the statistical analysis of inputs, clustering of inputs

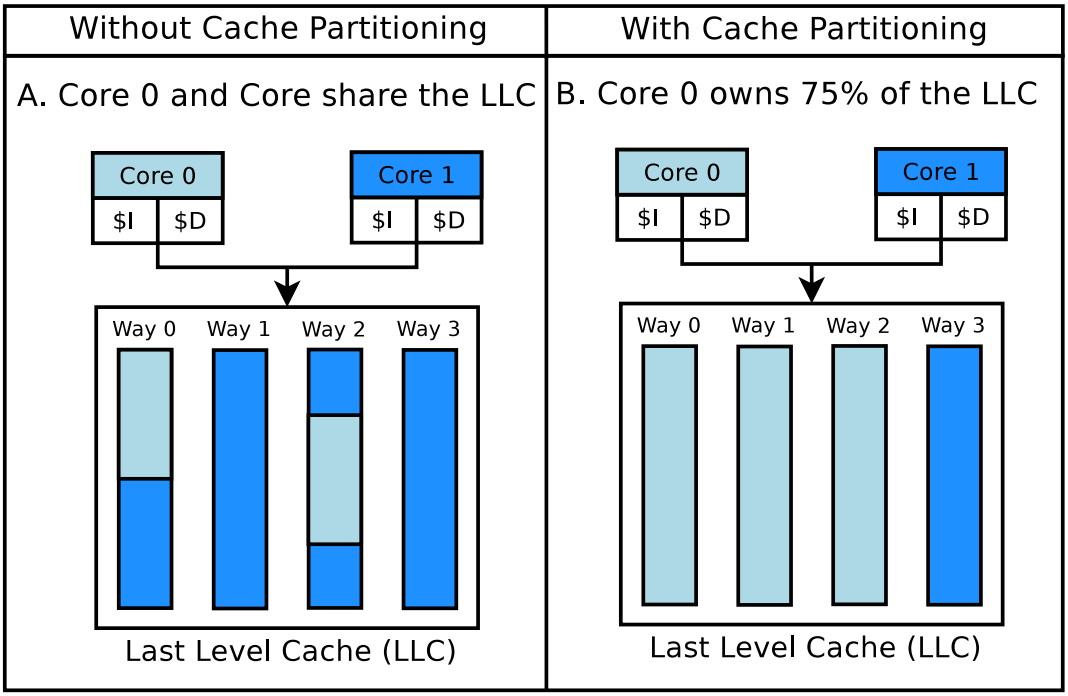


Figure 2.4: Example showing the cache partitioning at various proportions.

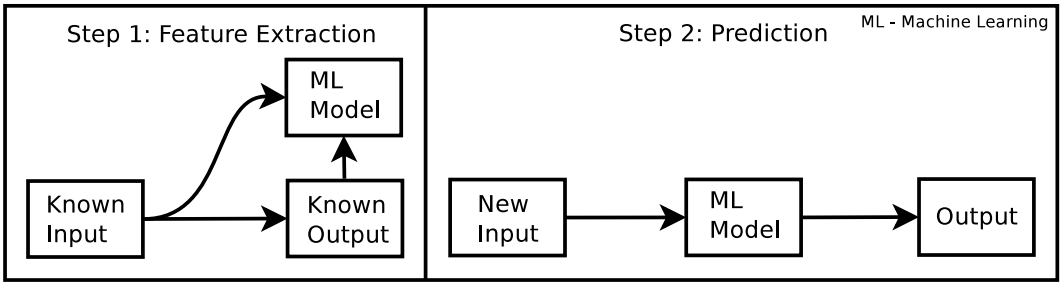


Figure 2.5: Formation and working of a supervised learning model.

is carried out. This clustering is based on the number of such clusters required, which is given as an input to the model.

Supervised Learning is used in this thesis, describing more about unsupervised learning is beyond the scope of this thesis.

2.5.1 Linear Regression Model

Linear regression tries to extract the linear relationship that exists between the input and output. It is expressed as weighted sum, whose weights β are determined to minimise the squared error between the predicted output Y_P and actual output Y_R. In other words, a linear combination of the input is used to predict the output. The linear relationship is given in equation 2.1.

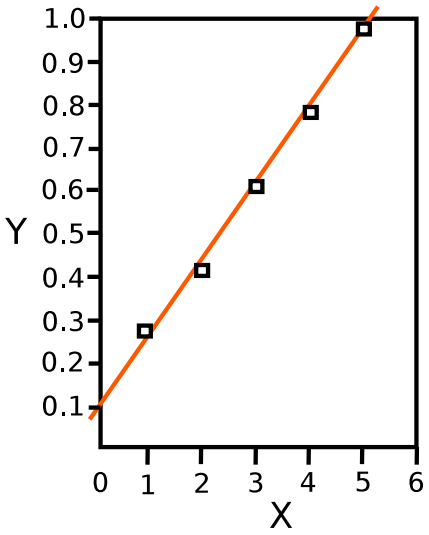


Figure 2.6: Example showing the linear regression. Assuming β_0 is 0.10 and β_1 is 0.17. The output Y is defined as $Y = \beta_0 + \beta_1 \times X$.

$$Y_P = \beta_0 + \beta_1 \times X_{,1} + \cdots + \beta_n \times X_{,n} \tag{2.1}$$

Figure 2.6 shows a red color line. This line is called a linearity line. This line tries to connect all the small square boxes. A small square box represent the actual output(Y) obtained from the given input(X). When the points are connected, lesser the distance between the square box and the line, lesser is the error between the predicted and actual output.

2.5.2 Decision Tree Model

Decision trees use a model of decisions and their consequences (like an If-Then-Else statement in a C language). When compared to linear regression, which requires complex multiplication of weights with the observed inputs, decision tree has less hardware overhead, by having only comparators.

Figure 2.7 shows an example of decisions involved in selecting the number of cache ways as one, two, four and eight ways. First, the input(X) is compared against a threshold value of 10, depending on the outcome, the tree branches left or right. The comparison of the input(X) continues with the threshold values of 100 and 1. Finally, the number of cache ways is determined as output(Y). These threshold values are determined to lower the error between the predicted and actual output. Thus the key factor depends on determining the threshold value, by reducing the error in the output(Y) for

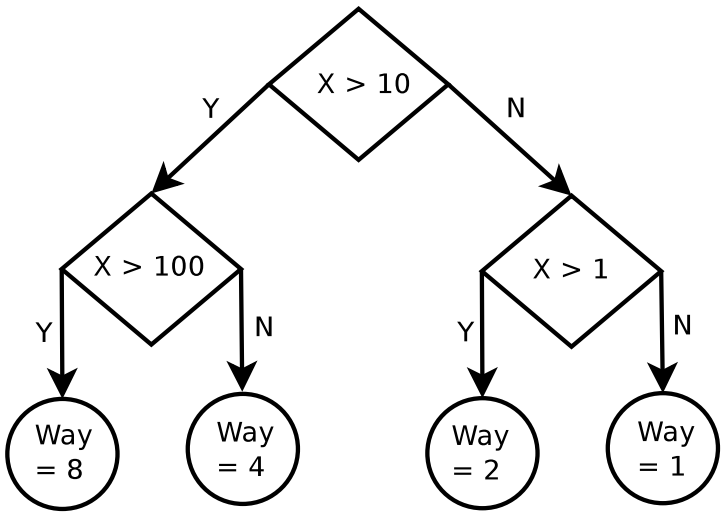


Figure 2.7: Example showing the Decision tree.

the given input(X).

Depending on the complexity of the relationship that exist between the input(X) and output(Y), a choice of either a linear regression or a decision tree can be used. In Section 4.6.2, a more detailed analysis of the output between decision tree and linear regression model have been presented.

2.5.3 Evaluation Methodology

An important aspect of any machine learning technique is its validation. In supervised learning, the machine learning model is built from the known inputs. Hence, care should be taken to separate the known inputs from the new inputs. Leave-one-out cross-validation is the standard machine learning evaluation methodology. This is followed when the training data is small. Consider the input set has N data points, the training data is formed from N-1 data points. Once the machine learning model is built, it predicts the N^{th} data point that has been left out.

2.6 Summary

This chapter has presented the basics of cache terminology and the parameters that are used in cache reconfiguration. A brief introduction to cache reconfiguration, cache partitioning and machine learning models have been presented and this forms the basic background for the Chapters 4, 5 and 6. The next chapter presents the literature survey of different cache management techniques targeting both power and performance.

Chapter 3

Related Work

This chapter discusses the research conducted on cache management that is relevant to this thesis. First, it presents the related work on improving the performance of the processor through better replacement policies, good hashing function and pre-fetching. Second, it gives an overview of the power saving techniques, both dynamic and static power in caches. Third, it presents the related work in reconfigurable caches present in single core processor. Finally, it presents the related work in cache partitioning for multi and many core processors.

3.1 Cache Management

Multi-level caches have been used to reduce the miss penalty. Cache hierarchies have been employed to bridge the performance gap between the processor and memory. In such a system, there will be a private Level-1 cache and a shared Last Level Cache (LLC). Thus a private and a shared cache have to be collectively managed for the better utilization of caches.

3.1.1 Replacement Policies

Identifying the right victim for replacement becomes more difficult as the number of cores in a CMP increases. Caches typically use Least Recently Used (LRU) replacement policy or some approximation of LRU, known as Pseudo LRU (PLRU) policy [7]. A good replacement policy should aim at reducing the number of misses by selecting the victim that will be accessed farthest into the future as shown in [11], which also provides a theoretical upper limit on performance achieved over LRU.

In a multi-core system, the baseline LRU policy gives more priority to a core that makes frequent requests to the shared cache. Therefore it serves the requesting core based on the demand. Several proposals have been made to overcome the shortcomings of LRU [68, 85, 101, 115]. Quereshi et al [108, 66] proposed an adaptive replacement policy that adapts according to the workload combinations that runs on different cores. Jaleel et al [67] proposed a replacement policy, that helps an operating system to make an efficient scheduling decisions.

3.1.2 Improving The Hit Rate

Direct mapped caches have a fast hit time compared to the set-associative caches, due to the restricted look-ups in tag and data. However they incur more conflict misses compared to a set-associative cache, because of various addresses competing for the same cache line. This creates a non-uniformity in cache accesses, due to some cache lines being accessed more heavily than others. Several proposals have been made to overcome this shortcoming in direct mapped caches [6, 10, 72, 119, 140, 46].

The hash-rehash [5] cache, attempts to access the direct mapped Level-1 cache using a hash function, upon a miss, it re-hashes using a different hashing function and this continues until the required data is found or the number of misses reaches a predetermined threshold count, then it accesses the next level cache. Thus they try to spread the cache accesses uniformly across the cache lines by changing the hashing function .

A Victim cache [71] is a small, yet fully associative buffer that holds the evicted entries from the heavily accessed cache lines of the level-1 direct mapped cache. Upon a miss, entries in the victim cache are checked to see a hit. When there is no match in the victim cache, the next level cache is accessed.

The Line distillation [110] technique improves the cache usage by discarding the unused words which is done by tracking the word usage information of a cache line until it reaches a predefined recency position in the LRU stack and then evicts the words that have not been used.

The following have been proposed to improve the hit rate of set-associative caches. Different hashing functions have been used to spread the cache accesses across the set-associative caches.

Skewed associative caches [120] improves the hit ratio of set-associative caches by using a different hashing function for different banks. A two-way set associative

skewed cache gives the hit rate of a four-way set associative cache.

Variable Way Set Associative (V-Way) [111] cache offers a variable associativity on a per cache set basis, by increasing the tag-store entries relative to data-store entries. They target the second level cache by offering a global replacement decision of a fully associative cache while maintaining the hit latency of a set associative cache.

The Z-Cache [117] offers the benefits of skewed associative caches for cache hits, Upon a cache miss, it performs cache replacement over multiple steps. First it identifies the right candidate for replacement and then it performs several relocations to accommodate the new incoming block. The downside to this is the bandwidth utilization during relocations.

3.1.3 Cache Pre-fetching

In cache pre-fetching, as the name implies, a cache line is fetched into the cache, before a read/write request is made to the line. There are several research proposals on cache pre-fetching [22, 26, 29, 38]. The key idea of the pre-fetching scheme is to determine the data access pattern in the cache and then to pre-fetch the data accordingly.

There are several advantages to this pre-fetching when the data access pattern is correctly identified. Numerical programs often have a regular access pattern. Hence, pre-fetching will be very beneficial for improved performance. However, integer programs usually exhibit irregular access patterns. Such programs suffer performance degradation from pre-fetching, as the pre-fetching pollutes the cache with data that will not be referenced in the near future. Pre-fetching in this case, causes bandwidth contention and results in reduced performance. However, cache pre-fetching is orthogonal to this thesis.

3.1.4 Quality Of Service (QoS) In LLC

There have been various schemes to guarantee quality of service by assigning priority levels to threads and partitioning accordingly [78, 17, 44, 63, 64, 98].

Bitirgen et al. [14] applied machine learning to efficiently manage the shared cache and off-chip bandwidth for improved performance. They used dynamic voltage scaling to set the optimal per-core voltage level for various configurations.

Jahre and Natvig [65] studied the variation of the number of available Miss Status Holding Register (MSHR) and their impact on the total miss bandwidth available to each thread running on a CMP.

Ebrahimi et al [33] improved [65] by proposing a scheme that offers fairness in Cycles Per Instruction (CPI) among co-executing applications. This is achieved by throttling the Miss Status Holding Register (MSHR). When an application generates more requests to the LLC, an online feedback mechanism throttles the requests depending on the co-executing application's cache requirement.

Herdrich et al. [50] proposed a clock modulation technique to avoid the interference caused by a lower-priority application over a higher-priority application in CMP systems.

Spill-receive cache [107, 116] offers QoS by a spill-receive mechanism for the private LLC's present in CMP. The capacity of a private LLC is increased or decreased by either spilling or receiving the cache contents to or from other private LLCs respectively.

3.2 Low Power Caches

The power, is a function of both dynamic and static power. The power consumption of a set-associative cache tends to be higher than that of a direct mapped cache, due to the activation of available cache ways upon every cache access. This includes the simultaneous activation of both the tag and the data parts of the cache. Power dissipation in caches has been heavily researched and several proposals made to reduce both dynamic and static power in caches.

$$Total_{Power} = Power_{Dynamic} + Power_{Static} \tag{3.1}$$

3.2.1 Dynamic Power

Dynamic power is due to the switching activity that takes place in transistors. It is proportional to the product of the number of switching transistors and the switching rate. It is calculated for every cache access. The following formula gives the dynamic power.

$$Power_{Dynamic} = 0.5 \times Capacitive\ Load \times Voltage^2 \times Frequency\ switched \tag{3.2}$$

Dynamic power is proportional to the product of load capacitance of the transistor, the square of voltage and the frequency of switching. Thus, reducing the switching activity, reduces the dynamic power consumption.

The following proposals have been made to reduce the dynamic energy consumption in caches.

Filter cache [79] is a small direct mapped cache, typically a level-0 cache, present before the level-1 cache. As the name implies, it filters out the frequently accessed data present in level-1 cache. If most of the level-1 requests are serviced by the filter cache, then most of the level-1 cache switching activity can be greatly reduced. Thus, it provides a fast hit time and also reduces significant dynamic energy consumption of the level-1 caches.

The Data part of the cache accounts for more size in bytes compared to the tag part of the cache. Thus, it consumes more dynamic energy compared to the tag access. The phased cache [48, 76] divides the cache access into two phases. First, all the tags in the set are examined in parallel and no data accesses occur during this phase. Second, if there is a hit, then a data access is performed for the hit way. Hence a phased look-up will reduce the dynamic power on every cache access. For the first level cache, whose hit time is in the critical path, this will incur an additional delay of one cycle hit latency. Therefore the scheme is beneficial in lower level caches but not in first level cache, where a cache hit time is in the critical path. Hence this scheme is used to reduce the dynamic energy in lower level caches.

Way-Prediction [58, 92, 16, 104], on the other hand, reduces dynamic energy by predicting exactly one way on every cache access. On a prediction hit, only one way is accessed, resulting in a fast tag and data look-up along with the reduction in dynamic energy. On a prediction miss, the cache access time is increased by one cycle latency, as it needs to check all the tags and data in the next cycle. If there is a cache hit, the request is serviced immediately, if not the next lower level cache is accessed. Thus, during a prediction miss, it behaves like a normal set associative cache. Way prediction is beneficial when the cache access is predictable, such as in instruction cache. For the data cache and the next lower level cache, whose cache access is irregular and unpredictable, way prediction incurs more latency for miss prediction. Since the instruction cache is accessed more frequently, more dynamic energy saving is achieved.

Way guarding [39], uses counting bloom filters to determine whether an incoming address is present in the cache or not. If there is a hit in the entries of the counting bloom filters, then several cache ways that might contain the requested address are enabled simultaneously. If there is a miss in the entries, then it is guaranteed that the requested address will not be present in the cache. This has an advantage over way prediction, as it does not incur extra look-ups on a miss. On a hit, way guarding enables

more cache ways than way prediction. By hashing the incoming address, storing it in the bloom filter and updating the entries on every cache miss, way guarding tries to remember the addresses that are cached in the cache. Thus, Way guarding can be used over way prediction to reduce dynamic energy consumption in all levels of caches.

Selective cache ways [8] have been proposed to selectively enable the cache ways based on the cache requirement of an application. Some applications will use the full capacity of the available cache, while some will use a part of the available cache. Enabling or disabling the cache ways based on the requirement of an application, offers reduction in dynamic energy, whilst maintaining high performance. The cache way enable signals can be controlled through software or can be implemented via specialized instructions.

3.2.2 Static Power

Static power, also commonly known as leakage power, is due to the leakage current that flows even when the transistor is turned-off. Static power is proportional to the transistor count. It is calculated every cycle. The following formula gives the static power;

$$Power_{Static} = Voltage \times N \times k \times Current_{Static} \tag{3.3}$$

N is the number of transistors, k is a design dependent parameter. Hence, static power is proportional to the number of transistors and it increases as the transistor size reduces. The following are some of the proposals to reduce the static energy in caches.

Drowsy cache [36] keeps the cache line in a low power state, known as the drowsy state. When a cache line is accessed, its drain voltage V_{dd} should be high or at value 1 for the contents to be preserved. By lowering the V_{dd} to a predetermined threshold value say for example $0.7V_{dd}$, the contents are still preserved. When V_{dd} is further lowered beyond the threshold value, the cache contents are lost. The key challenge in designing the drowsy cache is to determine the threshold value. This is the value at which all the cache lines, irrespective of their irregularities in fabrication, should able to preserve the cache contents, while being in the lowest V_{dd} for the maximum static energy reduction. When a cache line is in the drowsy state, its contents cannot be read/write, it requires one cycle to activate or to bring the cache lines to V_{dd} high, before the read/written action. Thus it incurs one cycle extra hit latency for activating and then accessing the cache line. First level caches have short hit time constraints. Hence drowsy caches can be implemented in the lower level caches, whose hit time is

not in the critical path and also these lower level caches are larger than the first level caches, providing more room for energy saving whilst maintaining high performance. As the CMOS technology advances, static energy will be more dominant than dynamic energy, making drowsy caches increasingly relevant.

The Gated-V_{dd} [105] technique, drains out the V_{dd} completely, making the cache lines lose their contents. This scheme operates at $0V_{dd}$ compared to the drowsy scheme, offering more static energy reductions. The hardware overhead increases, with a gating transistor added for every cache line. Thus connecting several cache lines to the gating transistor, simplifies the power gating scheme. Powering down the cache lines can be carried out at a bank level, rather than at line granularity, making the scheme more coarse grain. The power down signal can be connected to all the gated V_{dd} transistors, which when enabled drains out the power completely in a bank, resulting in turning-off the entire bank. As described in [8], cache ways can be selectively enabled/disabled based on the application's requirement. Gated V_{dd} can be combined with the selective cache ways to turn-off the disabled cache ways, offering both dynamic and static energy savings.

Kaxiras et al. [74] developed cache decay which is a state-destroying low power scheme.

Meng et al. [90] explored the upper limits of reducing leakage power by combining both drowsy and gated-Vdd techniques. However, this work is only a theoretical upper bound since it assumes the existence of an ideal pre-fetcher which is impossible to provide in practice.

Agarwal et al. [4] proposed a gated-ground scheme for turning off cache lines whilst preserving their contents. These kind of circuits require a single supply voltage.

3.3 Reconfigurable Single Core Cache Architecture

Reconfigurable caches are not new. Several researchers have investigated configurable cache designs by varying cache parameters such as the size, line size and associativity. The state-of-the-art reconfigurable cache architectures can be grouped into the following categories.

3.3.1 Set-Only Reconfiguration

In a set-only cache, the cache size is increased and decreased by enabling or disabling one or more sets respectively [139]. At smaller cache sizes, unused sets can be turned off to reduce the static energy consumption [105]. The miss ratio was used by Yang et al. [139] to guide cache reconfiguration, varying the size by masking index bits through a shifting operation. This allowed them to alter the cache size one factor of this step at a time.

3.3.2 Way-Only Reconfiguration

Albonesi [8, 9] proposed a cache design that can vary size and associativity by enabling or disabling cache ways, saving dynamic power when using less resources. This is a coarse-grained reconfiguration approach that may increase capacity and conflict misses [139].

Zhang et al. [143] proposed way-concatenation to reduce dynamic power by accessing fewer ways, depending on the associativity. This was performed once, before an application started execution. They also used way-shutdown to decrease cache size by turning-off unused ways using the Gated-V_{dd} method [105]. However, they did not address the changes required to the control signals when adding in way-shutdown.

Later, Ross et al. [41] described an extension to enable dynamic cache reconfiguration. However, they do not describe the control signals required to combine way-concatenation with way-shutdown. Furthermore, this requires flushing of dirty data to the next level cache when increasing associativity for a fixed cache size.

3.3.3 Set-and-Way Reconfiguration

Yang et al. [138] combined configurable set [105] and way [8] architectures to offer a hybrid cache that gives flexibility in terms of size and associativity. Increasing associativity by adding ways but keeping the cache size fixed, results in a copying back of previously stored data. This motivates the need for a new cache architecture that supports dynamic reconfiguration without incurring extra cycles for copying back information, while increasing the associativity.

Chapter 4 introduces a novel technique that combines way and set reconfiguration, and delivers improved energy savings. The proposed scheme is complementary to heterogeneous way-sizes [3], concatenating lines [142], wide-tag partitioning [21], dual

data cache [40] and Pseudo Set-Associative Cache [56].

3.3.4 Other Approaches

Apart from caches, other hardware micro-architectural adaptations include, issue width [57, 60, 130], issue queue [37, 1, 30, 15], pipeline [34], re-order buffer and register files [1, 30]. Similarly, a software controlled adaptation is also explored by [134, 54, 62].

Micro-architecture design space exploration (including for caches) has been studied by several researchers using run-time [25, 14], linear regression models [69], table-driven models [80], analytical models [73, 100], dependence graph [35], neural networks [32, 31, 59, 61], radial basis functions [70], and spline functions [83, 81, 82, 87, 84]. However, these proposals assume an abundant hardware resource availability and also none of these proposals addresses the question of how to achieve these benefits.

3.4 Cache Partitioning For Multi Core Processors

Existing state-of-the-art cache partitioning techniques can be split into two groups;

3.4.1 Partitioning For Performance

Cache partitioning has been widely studied in the past with both static and dynamic schemes proposed. Chiou et al. [23] were the first to introduce column caching and also proposed changes that are required by the replacement policy to be aware of partitioning. Suh et al. [125, 124] used the recency position of hits in cache lines to drive dynamic cache partitioning. However, a global monitoring scheme is used to collect data and hence hit statistics of individual applications get polluted by other co-executing programs.

Later Qureshi et al. [109] addressed these shortcomings with utility based cache partitioning (UCP) that uses a low-overhead auxiliary tag directory to monitor each core's cache usage through the LRU stack property [89]. An auxiliary tag contains 32 sets and have the associativity same as the LLC. These 32 sets capture the accesses going to the 32 sets of LLC, that are separated apart by an equal distance. For example, assume a LLC that has 2048 sets. All accesses going to set 0 of LLC will go to set 0 of auxiliary tag. Similarly, all access going to set 64 of LLC will go to set 1 of auxiliary tag and this continues upto set 2047 of LLC going to set 31 of auxiliary tag.

Since 2048/32 gives 64, 32 sets are sampled by an equal distance of 64. Cache utility curves are generated periodically and a look-ahead algorithm is used to determine the required partitioning decision. This thesis uses the cache monitoring scheme and a modified lookahead algorithm from UCP. However, our method for creating partitions and transferring blocks between cores is different to Qureshi's since we focus on energy efficiency.

Xie et al. [136] and Jaleel et al. [68] modified the shared cache replacement policy to provide performance benefits compared to an unmanaged cache. A two-dimensional cache partitioning was proposed by Chang et al. [20]. This allowed both space and time sharing within the cache, meaning that a few processors share a small cache region for particular time interval while the rest share the remaining large region.

Cooperative caching [19, 51, 52] combine the strengths of private and shared cache organizations by forming an aggregate shared cache through cooperation among private caches.

The thrasher caging scheme [137] identify workloads that thrash the cache and isolate them through partitioning. This technique obtain the benefits of partitioning for thrashing applications and an unmanaged cache for non-thrashing workloads, targeting performance. Similarly, Sanchez et al. [118] proposed fine-grained partitioning using an efficient hashing function. The scheme provides data isolation, with a small unpartitioned area that can be used by competing cores to increase their original partitions, rather than taking ways from other cores.

Lee et al. [86] proposed a cache management policy for CPU-GPU heterogeneous architecture. They use core-sampling mechanism to detect the caching effects of a GPGPU application.

There have been proposals to perform set-wise cache partitioning [113, 132, 93, 94, 95]. However, in a dynamic setting, these schemes would require frequent flushing of data due to the varying memory requirements of different phases of the programs.

Chandra et al [18] studied the impact of inter-thread interference by predicting the number of cache evictions that would be introduced by another thread that runs in a CMP system. However, fully-partitioned caches inherently avoid inter-thread interference. Static schemes have determined the optimal partitions for any combination of applications [123] or have been used to set parameters for various management policies [55].

Chapter 5 introduces Cooperative Partitioning, a novel cache partitioning scheme that is orthogonal to [23, 125, 124, 109, 19, 51, 52, 86] and can be applied to these

schemes to offer energy reduction on top of performance benefits.

3.4.2 Partitioning For Energy Efficiency

In terms of energy efficiency, Reddy et al. [114] statically profiled each application to determine their cache requirements. This information was used to compute cache partitions that can be adapted to sets and associativity. However, as the number of workload combination increases, static profiling becomes more impractical.

Albonesi [8] proposed a cache design that can vary its size and associativity by enabling or disabling cache ways. Powell et al. [105] developed a gated-Vdd (non-state preserving) technique to reconfigure the cache and turn off unused cache lines. Meng et al. [90] explored the upper limits of reducing leakage power by combining both drowsy [36] and gated-Vdd techniques. However, this work is only a theoretical upper bound on energy saving since it assumes the existence of an ideal pre-fetcher, which is impossible to provide in practice.

Abella et al. [2] studied the possible ways to combine cache modules with different voltages for high-performance and low power.

Finally, Kedzierski et al. [75] proposed a power-aware partitioning using a drowsy cache implementation to reduce both dynamic and static power.

The Chapter 5 introduces Cooperative Partitioning, a new technique that provides both dynamic and static energy savings and the drowsy scheme can also be implemented in the proposed cache to offer further energy reductions.

3.5 Cache Partitioning For Many Core Processor

There is a rich body of work in the literature concerning cache partitioning, especially in the recent past. All the previously proposed cache partitioning schemes work better when the number of core to way ratio is 1:4 or some thing similar [109, 136, 118, 129].

Jaleel et al. [68] obtain performance benefits in a shared cache by modifying the replacement policy. The more dynamic replacement policy, DRRIP [68], which using set duelling monitors to keep track of competing static replacement policies and dynamically select the one with the lowest number of misses, after a fixed number of cycles. In a similar vein, Jaleel et al. [67] proposed a scheduling aware cache replacement algorithm. This scheme requires two or more LLCs and more cache ways than cores in the system.

Applications that thrash the cache can be detrimental to the performance of other workloads and the thrasher caging scheme [137] identify and isolate these programs through partitioning. This allows non-thrashing, co-habiting workloads to obtain the benefits of an unmanaged cache, while not affecting the performance of the thrashing applications.

Muralidhara et al. [97] proposed cache partitioning for parallel applications targeting performance. Their scheme monitors each thread's cache requirements and allocates ways to each based on this. However this does not distinguish between private and shared data, missing significant opportunities for energy saving.

The D-NUCA [77, 24] scheme migrate the cache lines towards the core that frequently accesses it, albeit within the same level of cache. This reduces the latency of frequently accessed cache lines.

The R-NUCA [47] scheme extends D-NUCA to identify private and shared data at a page level. This enables data to be moved close to the cores that request it most often, increasing performance from a non-uniform cache architecture. Cuesta et al. [27] also identified shared and private data at a page level, bypassing the directory lookup for private data.

Chapter 6 introduces Region Aware Cache Partitioning (RECAP), a novel cache partitioning technique for many-core processors. The proposed approach is orthogonal to [97, 77, 24, 77, 47, 27, 67, 68] as it can be used whatever the core-to-way ratio is and it also realizes both static and dynamic energy savings, while using significantly less hardware.

3.6 Summary

This chapter has presented prior work on cache management that acts as a foundation for designing next generation caches in many core processors. A detailed study of prior work on designing low power caches that helps in the design of both energy efficient and high performance caches that will be present in future systems. The next chapter discusses an energy efficient cache architecture for a single core system.

Chapter 4

Smart Cache Architecture

The demand for low-power embedded systems require designers to tune processor parameters to avoid excessive energy wastage. Tuning on a per-application or per-application-phase basis allows a greater saving in energy consumption without a noticeable degradation in performance. On-chip caches often consume a significant fraction of the total energy budget and are therefore prime candidates for adaptation.

This chapter presents a Set and way Management cache Architecture for Run-Time reconfiguration (SMART cache), a cache architecture that allows reconfiguration in both its size and associativity. The evaluation demonstrates that the energy-delay of the Smart cache is on average 70% and 12% better than the baseline configuration for a two-core and four-core system respectively, and just 2% away from the oracle result and also with an overall performance degradation of less than 2% compared with a baseline statically-configured cache.

4.1 Introduction

Cache memories contain a large number of transistors and consume a large amount of energy. For instance, 60% of the StrongARM's area is devoted to caches [88]. For this reason many processors, particularly intellectual property cores, allow the configuration of the caches to be determined at design time, according to the requirements of the target applications. Customization of cache parameters may be static or dynamic; in a static approach the designer sets the cache parameters before synthesis, whereas in a dynamic scheme the cache parameters can be modified within a certain range at run-time.

When cache parameters are determined statically, a single configuration is chosen

by the designer to trade-off performance against energy consumption. Static configurations require less on-chip logic, validation and testing than performing dynamic reconfiguration. However, they do not have the ability to react to changes in cache requirements both across programs and within the same application. The hypothesis is that, in order to achieve optimum energy efficiency, cache parameters should be reconfigured at run-time in response to the changing requirements of the running application.

Dynamic cache reconfiguration is not a new topic, having been previously studied by a variety of researchers [3, 8, 21, 41, 42, 43, 105, 138, 141]. These schemes monitor the miss ratio at run-time, reconfiguring the cache whenever it reaches a certain threshold value. However, they are limited in the amount of flexibility they provide — either performing set-only or way-only reconfiguration — or they consult larger sub-banks on each access than are actually required. Furthermore, relying solely on the miss ratio to determine the correct time to reconfigure does not always give a good indication of the changing requirements of the application.

This chapter proposes configurable cache architecture, called the Smart cache that allows reconfiguration of both the size and associativity of Level-1 instruction, data and Level-2 caches, providing maximum flexibility to the application

This chapter makes the following contributions:

- First it proposes a configurable cache architecture that allows reconfiguration of both the size and associativity of each cache, providing maximum flexibility to the application. The Smart cache is compared against state-of-the-art cache reconfiguration techniques by implementing state-of-the-art scheme in the simulation infrastructure described in Section 4.4 and result shows that Smart cache energy-delay product is on average 18% better than the state-of-the-art across our benchmark suite.

- To demonstrate the performance of the proposed scheme, a decision tree model is developed to monitor the behaviour of each cache and dynamically reconfigures in response to changing application requirements. The proposed scheme demonstrates that the new approach causes negligible performance loss, yet achieves an energy-delay product improvement of 0.17 and 0.34 in the data cache and level-2 cache respectively.

- The proposed scheme has been extended to multicore systems and we have evaluated both two-core and four-core systems.

- Prediction accuracy of the decision tree model is compared against the linear regression model.

- Finally, this chapter has evaluated the decision tree model on a new benchmark suite (namely SPEC CPU 2006), after having been trained on the SPEC CPU 2000 workloads.

The rest of this chapter is structured as follows. Section 4.2 describes the Smart cache and section 4.3 presents the decision tree model with the features of cache behaviour that it monitors. It also discusses the power and performance overheads of the proposed approach. Section 4.4 describes the experimental set-up and section 4.5 presents the results. Finally, section 4.8 concludes.

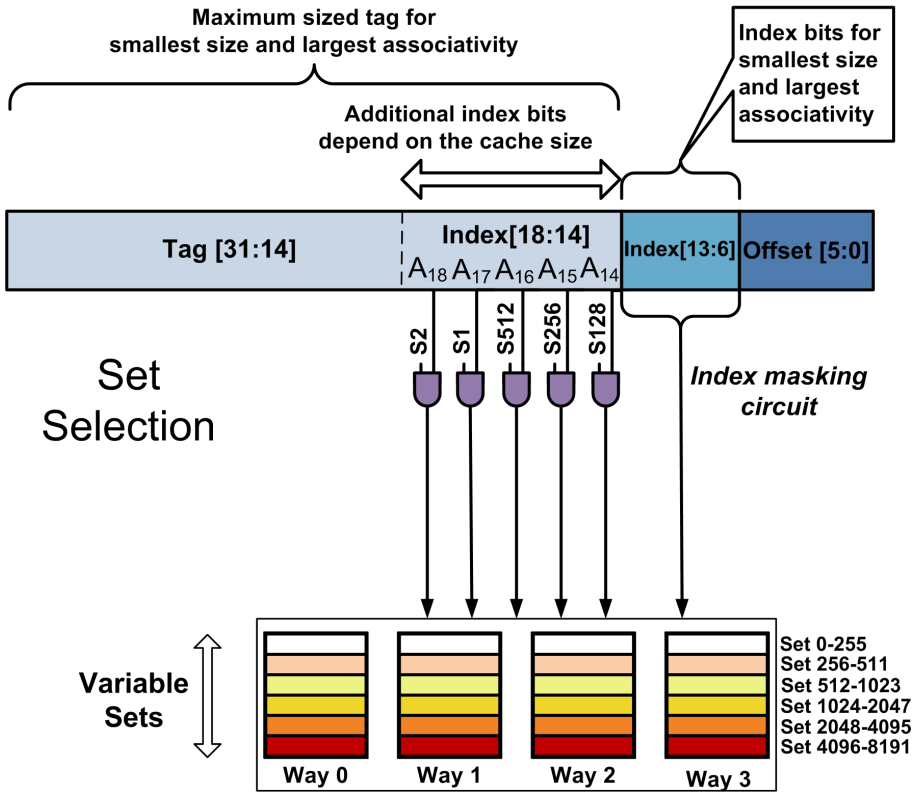
4.2 The Smart Cache Architecture

This section describes the Smart architecture that is used for each cache within the system. Figure 4.1 shows how each address is mapped into the cache for a 2MB level-2 cache. There are two complementary circuits used in parallel that perform the mapping of the addresses, allowing the address to be routed to the correct set and way.

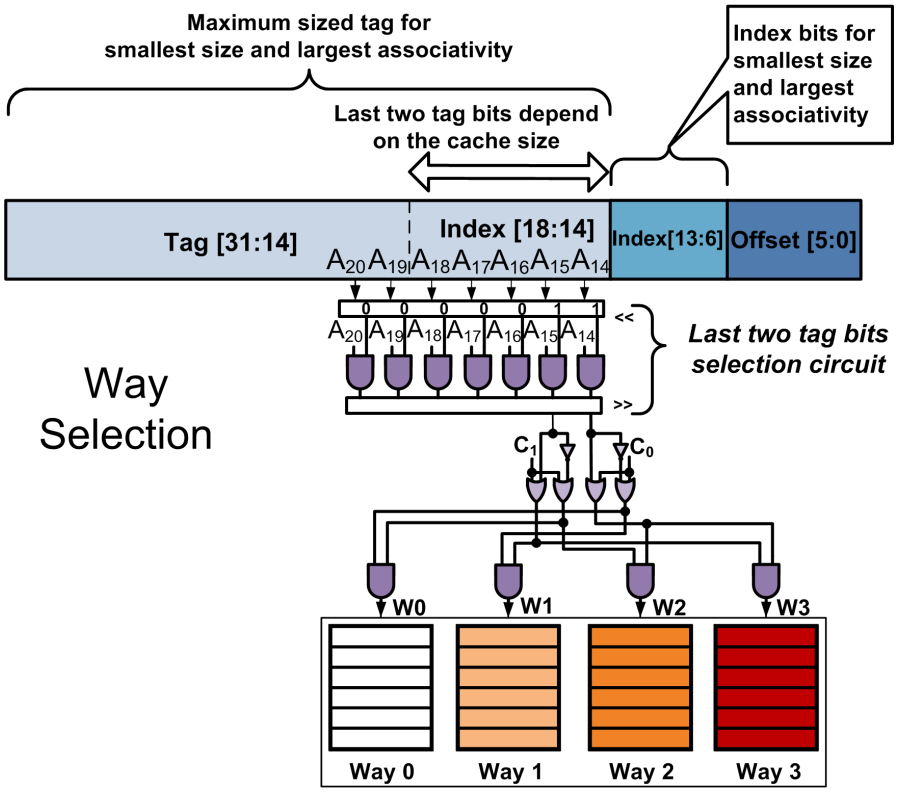
As the cache size and associativity varies, so does the number of bits needed for the tags. Smart cache architecture stores the maximum sized tag for each line (i.e., for the smallest cache and largest associativity). This section now describes how the address is routed to the sets and ways, then discuss the overheads of the architecture.

4.2.1 Set Selection

The sets are grouped in each bank by augmenting the cache with size selection bits that determine the sets that are enabled. These are then *AND*ed with bits from the index to determine the sets to access. In the 2MB level-2 cache shown in Figure 4.1, Size selection bits S128, S256, S512, S1, and S2 represents cache sizes of 128KB (sets 0-511), 256KB (sets 0-1023), 512KB (sets 0-2047), 1MB (sets 0-4095) and 2MB (sets 0-8191) respectively. A 64KB cache (sets 0-255) is always enabled even when all size selection bits are 0. The size selection bits could be set via a hardware scheme, or exposed to the software.



(a) Set Selection



(b) Way Selection

Figure 4.1: Organization of the Smart cache architecture. Varying the cache size (through the set selection circuits) is performed in parallel with altering the associativity (through the way selection logic).

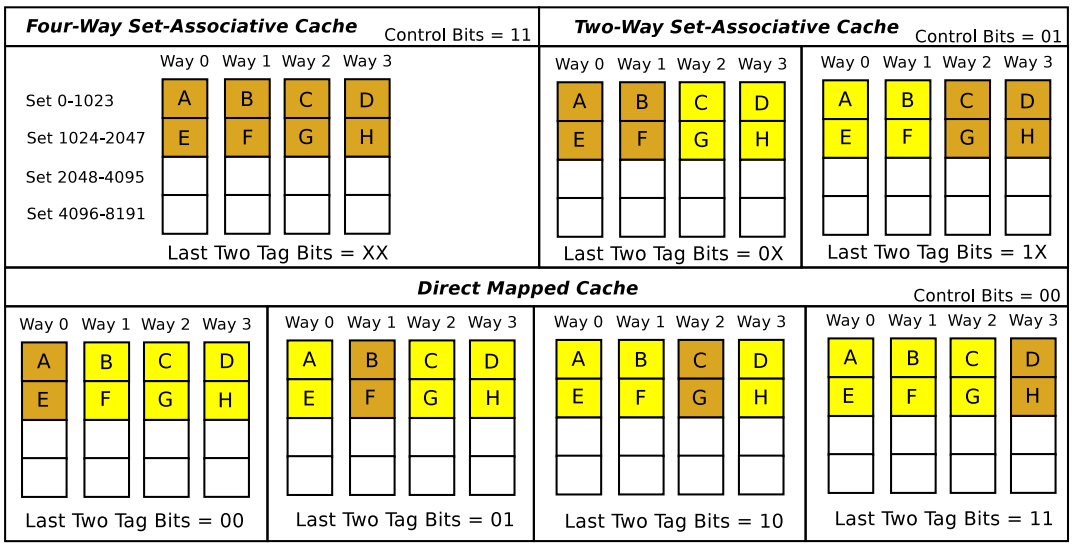


Figure 4.2: Working of the Smart cache. Brown, yellow and white regions show accessed, unaccessed and disabled sets respectively. Control bits determine the associativity and the last two tag bits determine the ways.

4.2.2 Way Selection

In order to control the associativity of the cache, a way selection circuit is used. This uses the last two tag bits and the size selection bits to route accesses to the ways that are enabled. Since the cache size can vary, the size selection bits are required to correctly identify the last two tag bits. In figure 4.1, for a small cache, they are represented by bits [15:14] and for a large cache, they are represented by bits [20:19]. Two control bits (C_0 and C_1) determine the ways that will be eventually accessed. For one-way associativity, any one of the way selection control signals W0, W1, W2 or W3 will be active. For two-way associativity, any two of the way selection control signals will be active. Finally, for four-way associativity, all way selection control signals are active. Table 4.1 shows how the control and tag bits map to the ways that are enabled.

4.2.3 Example Cache Access

Figure 4.2 shows how the control and tag bits map to the ways that are enabled in the Smart cache. As an example of how the set and way selection circuits work in tandem, consider a 512KB, two-way associative cache. In this scenario, size selection bits S128, S256 and S512 are set to 1, all others are set to 0. For the control bits, C_0 is set to 0 and C_1 is set to 1.

The address is routed to cache banks after passing through the selection circuits.

For all the cache configurations, the tag bits [31:14] are used. Depending on the cache size, the way selection circuit selects two tag bits, in this example it uses bits [18:17] which corresponds to a cache size of 512KB. Assuming that they are both 1, then ways W2 and W3 are accessed. In the set selection circuit, bits [16:6] are also passed through with the index to select the correct lines from sets 0-2047. All other sets are turned off for static power saving.

4.2.4 Overheads

The way selection circuit does not appear in the cache's critical path because it can operate in parallel with the tag and data array address decoders [143]. The set selection circuit can be folded into the decoders to avoid any delay in calculating the index bits [105]. Therefore there is no increase in the cycle time for accessing the cache using the smart cache approach. However, after reconfiguring the cache to a smaller size, unused sets and ways are turned off, destroying their contents. Therefore dirty lines must be flushed back to the next level in the memory hierarchy. In all simulations, the flushing cost that includes both power and performance costs for copying back the dirty lines are added. Power and performance overheads of reconfiguration are discussed in detail in section 4.3.3.

The area overhead of the smart cache is 0.5% over the baseline. This is due to the extra control circuitry required to perform set selection, way selection and reconfiguration. This value has been obtained from a version of Cacti-5.3 [131] that has been modified to support the new circuitry.

4.2.5 Relation to Prior Work

There are several key differences between the Smart cache and state-of-the-art reconfiguration techniques. In the smart cache approach, the associativity and size are varied in parallel by using the way control signals and the size control registers. The Smart cache organizes ways at set boundaries as shown in Figure 4.2, which avoids flushing data back to memory when increasing the associativity but keeping the cache size fixed. This addresses the shortcomings of previous techniques [143], allowing dynamic reconfiguration of the cache. In addition to this, the Smart cache offers 3x more cache configurations than the set-only [139] and hybrid [143] schemes, which combine way-concatenation with way-shutdown.

Table 4.1: Mapping of tag and control bits to the active ways.

Associativity	Control Bits		Last Two	Active Way
	C_0	C_1	Tag Bits	Signals
One Way	0	0	00	W0
			01	W1
			10	W2
			11	W3
Two Way	0	1	0X	W0, W1
			1X	W2, W3
Four Way	1	1	XX	W0, W1, W2, W3

4.3 Controlling Reconfiguration

Having developed the cache architecture, this section now describes the method for dynamic reconfiguration. The cache behaviour is monitored by collecting statistics about the cache usage over a fixed interval size. These are then fed into a decision tree that computes the required cache size and associativity for the next interval. This section first presents the statistics used to characterize cache behaviour, then describe, the decision tree itself.

4.3.1 Cache Behaviour Characterization

In order to determine the best cache configuration to use for each program interval, the cache behaviour is monitored by gathering statistics about cache usage. These will accurately determine when the cache size or associativity needs to be altered. Two types of statistics are gathered: stack distance and dead set count.

4.3.1.1 Stack Distance

The stack distance [89] shows the position in a set's LRU chain that each access occurs in. This gives an approximation of the required associativity of the cache: if all accesses are in the MRU position, then the associativity can be reduced; if many accesses are in the LRU position or miss then the cache could benefit from higher associativity. This profiling is achieved by maintaining a counter for each position in the LRU chain for the whole cache to enable us to gather this information.

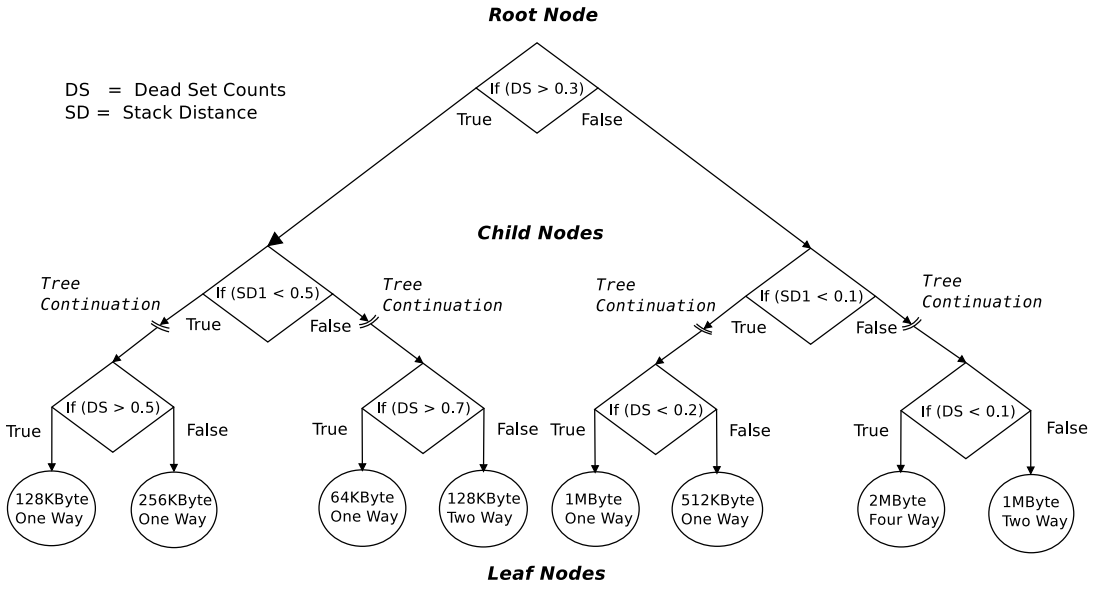


Figure 4.3: Example decision tree structure.

4.3.1.2 Dead Set Counts

A dead set is defined as one that is not accessed during a clearing interval (10K committed instructions). A large number of dead sets indicates that the cache could function adequately with a smaller number of sets (i.e., a smaller size). To monitor this, a 2-bit saturating counter is added to each set, clears them at the start of each clearing interval and increment them on each access. At the end of the each clearing interval, these counters are used to compute the total number of sets that have been accessed less than three times and are averaged over phase interval (10M committed instructions). This is a simple statistic, but it accurately identifies dead sets.

4.3.2 Decision Tree Model

Any form of dynamic hardware reconfiguration requires a decision-making process, driven by run-time measurements. This could be derived intuitively, but would then be open to the criticism that the model is trained specifically for the selected benchmarks. A different approach is taken by choosing machine learning to train a decision tree model, which clearly separates training and experimental data. An example is shown in figure 4.3. At each node in the tree, any one of the collected statistics is compared to a threshold value and depending on the outcome control passes either left or right to the corresponding child node. A decision tree is used to control reconfiguration because they can be easily implemented in hardware using a look-up table.

Assuming a dead set count (DS) of 0.6 and stack distance (SD) of 0.3, the example tree in figure 4.3 is followed to find the required configuration. The first comparison is performed in the root node where DS is compared to the threshold value of 0.3. Therefore the edge labelled *True* is taken and proceed down the left to the next node. This compares SD with 0.5, so the edge labelled *False* is taken. The final comparison considers whether DS is greater than 0.7, which is also *False*. Finally, the node containing the desired configuration, which is a 128KB cache with 2-way associativity is reached.

In order to determine the thresholds at each node, the decision tree needs to be trained using examples of good configurations from different programs. Training consists of finding thresholds that minimize the partition variance at each node. To do this, each training program is run on all cache configurations and the characterization statistics are gathered every interval. Good configurations are those that have an energy-delay lower than the baseline, with a maximum slowdown of 2% from the baseline across each interval. A leave-one-out cross-validation is used to train the decision tree using this data. This is a standard machine learning methodology and ensures the model is never trained on the benchmark it is tested on.

4.3.3 Overheads of Reconfiguration

There are two types of overhead that the smart cache dynamic reconfiguration scheme incurs. The first is power consumption and the second is performance.

4.3.3.1 Power Consumption

The power consumption of the statistics gathering logic is calculated for each cache in the processor and the models are used to drive reconfiguration. These have been incorporated into the simulator and the overheads are included in all results. The energy overheads of the statistics gathering logic are 0.01% of the baseline cache energy. The overhead of the decision tree model is 1% of the baseline cache energy consumption.

4.3.3.2 Performance

Traversing the decision tree to find the best cache configuration for the next interval takes several cycles. However, this is small in comparison to the time taken to run each interval. By halting the characterization shortly before the end of the interval, the decision tree traversal can be overlapped with the execution of the end of the interval,

hiding its latency. The performance overheads in actually performing reconfiguration of each cache is described in Section 4.2 and is included in all the results.

Altering the cache size or associativity may require dirty data to be written back to lower-level memory. When cache size is reduced by turning-off sets or ways, it requires dirty lines present in the future turned-off region to be written back to next lower level. When cache size is increased, blocks may map to different sets. This incurs extra misses for the first accesses to the new location and also requires dirty lines to be flushed back to the next level before increasing the size.

These reconfigurations incur extra cycles to copy back dirty lines to lower levels of memory, which in turn incurs extra performance and energy costs. However smart cache experimental results show that on an average reconfiguration is required once in every 10 intervals, or once every 100 million instructions. Thus, costs associated with this can be quantified by not varying cache size and associativity very often. The performance and energy costs of flushing these cache lines are included in all the results and, since reconfiguration is performed so infrequently, the overheads are small.

4.4 Experimental Setup

This section describes the simulator and benchmarks used to evaluate the smart cache reconfiguration approach.

The cache reconfiguration is implemented in the HotLeakage simulator [144]. The underlying power models are updated to use Cacti-5.3 [131] that has been modified to support the new circuitry for 70nm process technology. The simulator is altered to include the power and performance overheads of reconfiguring each cache, as previously described in Sections 4.2 and 4.3. Table 4.2 shows the configuration of the Alpha out-of-order superscalar, whose cache configurations are similar to an Intel Core 2 processor.

To evaluate the proposed technique, the SPEC CPU 2000 benchmark suites [122] as used as workloads, compiled with the highest optimization level. The *reference* inputs are used for running each application. Due to simulation time constraints and to maintain the continuity of cache behaviour, each workload is run from its start to 60 billion instructions. This ensures that the majority of each benchmark's behaviour are captured.

In the simulations, a phase interval of 10 million instructions is assumed. This

Table 4.2: Processor configuration.

Parameter	**Configurations**
Decode,Issue,Commit Width	4 ,4, 4
Register Update Unit Size	80
Load Store Queue Size	40
Instruction Cache Size	$1 \rightarrow 32$ KBytes
Instruction Cache Associativity	$1 \rightarrow 4$
Instruction Cache Line size	32 Bytes
Data Cache Size	$1 \rightarrow 32$ KBytes
Data Cache Associativity	$1 \rightarrow 4$
Data Cache Line size	32 Bytes
Level-2 Cache Size	$64 \rightarrow 2048$ KBytes
Level-2 Cache Associativity	$1 \rightarrow 8$
Level-2 Cache Line size	64 Bytes
Level-2 Cache Latency	6 Cycles
Memory access bus width	8 Bytes
Main-Memory Latency	97 Cycles
Technology	70nm

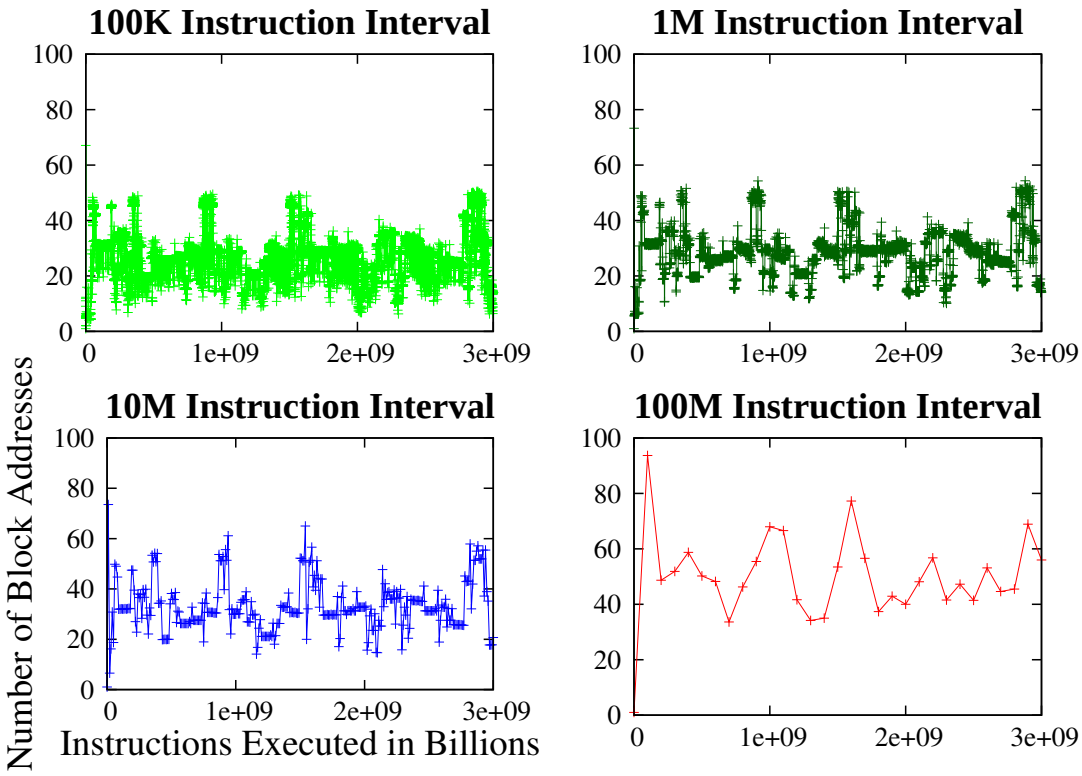


Figure 4.4: Varying number of block addresses in a bzip2 application, for different phase intervals are shown.

interval is selected after a characterization of the benchmarks from SPEC CPU 2000, using sampling intervals of 100K, 1M, 10M and 100M instructions and is shown in Figure 4.4. As the sampling phase interval increases, phase information is lost. Hence 10M interval is used and also this is a value commonly used by other researchers [121]. To gather data to train the decision tree model, each CPU 2000 benchmark is run on each cache configuration, gathering cache characterization statistics every interval. With 23 applications, 3 caches and 18 configurations for each, this totals 1,242 simulations. A leave-one-out cross-validation, a standard machine learning evaluation methodology is used to evaluate the proposed scheme, as described in Section 2.5.

The WEKA is used to analyse the training data-sets using data-mining algorithms [45]. WEKA comprises data classification, regression, clustering, association rules and visualization. It analyses, pre-processes and selects the key features from the training data-set and applies classification algorithms on the selected features.

4.5 Results

This section evaluates the Smart cache approach to dynamic cache reconfiguration. This shows the effects of dynamic reconfiguration using smart cache architecture and the decision tree model on each cache individually and a combined scheme for all caches at once. Subsequent sections analyse the Smart cache by performing a comparison with prior cache architectures, examining different predictors and benchmarks and considering multicore workloads.

In the graphs, the performance of the smart cache approach and the energy delay product achieved for the whole cache hierarchy is shown, taking into account reconfiguration and flushing costs. In addition to this, two comparison techniques are also shown. The first is the best static configuration of the cache, which corresponds to the configuration that has the lowest energy-delay and a maximum 2% performance loss across all benchmarks, with no dynamic reconfiguration. These are the same criteria used to select good configurations to train the decision tree. The second approach is an oracle which knows in advance the best configuration for each interval and incurs no overheads in dynamic reconfiguration. Although unrealistic in practice, this represents the lower bound on achievable energy-delay for any technique.

All energy-delay results are normalized to the baseline architecture which has an energy-delay value of 1.0. This is a processor where each cache is configured to its largest size and highest level of associativity.

4.5.1 Dynamic Cache Reconfiguration

This section evaluates the Smart cache architecture along with the decision tree model for dynamic reconfiguration of each cache in the hierarchy individually. In the following subsections, when reconfiguring the instruction cache, the energy spent in the baseline data and level-2 cache is added to the energy of the instruction cache. This is done to ease the comparison between the cache hierarchies. The same has also been employed for reconfiguring data and level-2 caches.

4.5.1.1 Instruction Cache

Figure 4.5 shows the performance and energy-delay of three different schemes when reconfiguring the instruction cache alone. As previously described, the first represents the best static configuration of the instruction cache across all benchmarks and the

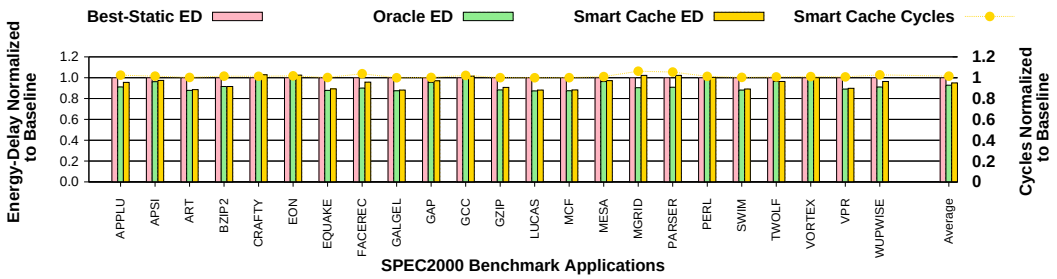


Figure 4.5: Performance and energy-delay characteristics of the instruction cache, while maintaining the data and level-2 caches at the baseline configuration. This chart shows the percentage reduction in total energy-delay achieved by reconfiguring only the instruction cache.

second is the oracle. The bars labelled *Smart cache* show the results from using the decision tree model along with the Smart architecture. The black circles show the performance achieved by the smart scheme, normalized to the baseline performance. The oracle and best static approaches never incur more than 2% performance loss, so their performance results have been excluded. For the instruction cache, the best static configuration is actually the 32KB cache with 4-way associativity (i.e., the baseline).

On average the energy-delay of the smart cache scheme is close to the theoretical maximum limit achieved by the oracle, with the difference being 2.2%. However, the smart cache loses 6% and 5% performance on *mgrid* and *parser* respectively, due to the decision tree model predicting a smaller cache configuration at the phase transitions. This is because the transition phase cache statistics for *mgrid* are similar to those from *applu* and *apsi* which need caches that are 2KB large, whereas *mgrid* requires a cache of 8KB. For the other applications, the decision tree model is effective at determining the correct cache configuration to use. Therefore, on average the smart cache incurs a performance loss of just 1.5% compared to the baseline, but achieve an energy-delay value of 0.95. A small performance loss such as this is expected since smart cache chose to bound performance losses to 2% of the baseline when identifying good configurations, as described in Section 4.3.2.

4.5.1.2 Data Cache

Turning the attention to the data cache, shown in figure 4.6, it can be seen that there is greater improvement to be gained than can be achieved from the instruction cache. Here the smart cache incurs a similar performance loss of 1.6% on an average, rising to 6.3% for *apsi*. This is again due to inaccuracies in the decision tree model that

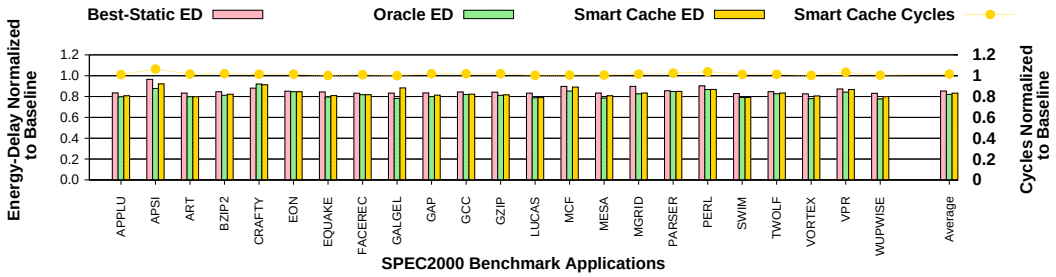


Figure 4.6: Performance and energy-delay characteristics of the data cache, while maintaining the instruction and level-2 caches at the baseline configuration. This chart shows the percentage reduction in total energy-delay achieved by reconfiguring only the data cache.

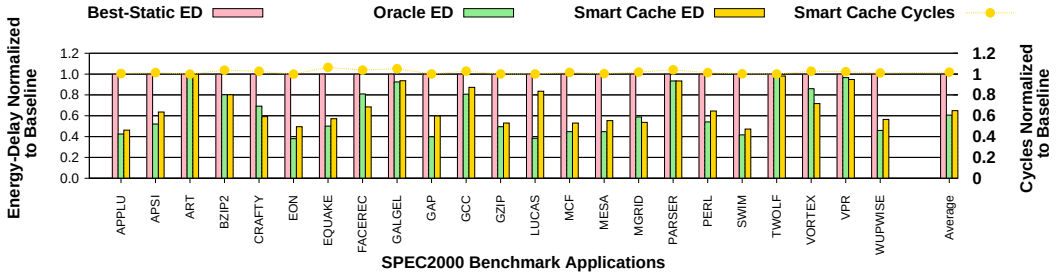


Figure 4.7: Performance and energy-delay characteristics of the level-2 cache, while maintaining the instruction and data caches at the baseline configuration. This chart shows the percentage reduction in total energy-delay achieved by reconfiguring only the level-2 cache.

infers a smaller cache configuration during phase transitions for this application than is actually required.

Considering the energy-delay, figure 4.6 shows that the smart cache again achieves results close to the oracle. The difference here is 1.1%. On an average the smart cache achieves an energy-delay value of 0.83. This is consistent across applications with *art*, *lucas*, *swim* and *wupwise* achieving values less than 0.8.

4.5.1.3 Level-2 Cache

Now consider the final cache in the hierarchy, which is the unified level-2 cache. Figure 4.7 shows the results of dynamically reconfiguring the level-2 cache. The best static cache configuration, across all benchmarks, is actually the baseline 2MB 8-way cache, so without dynamic reconfiguration, no energy savings are possible. Figure 4.8 and Figure 4.9 show how the cache configurations selected by the Smart cache compare

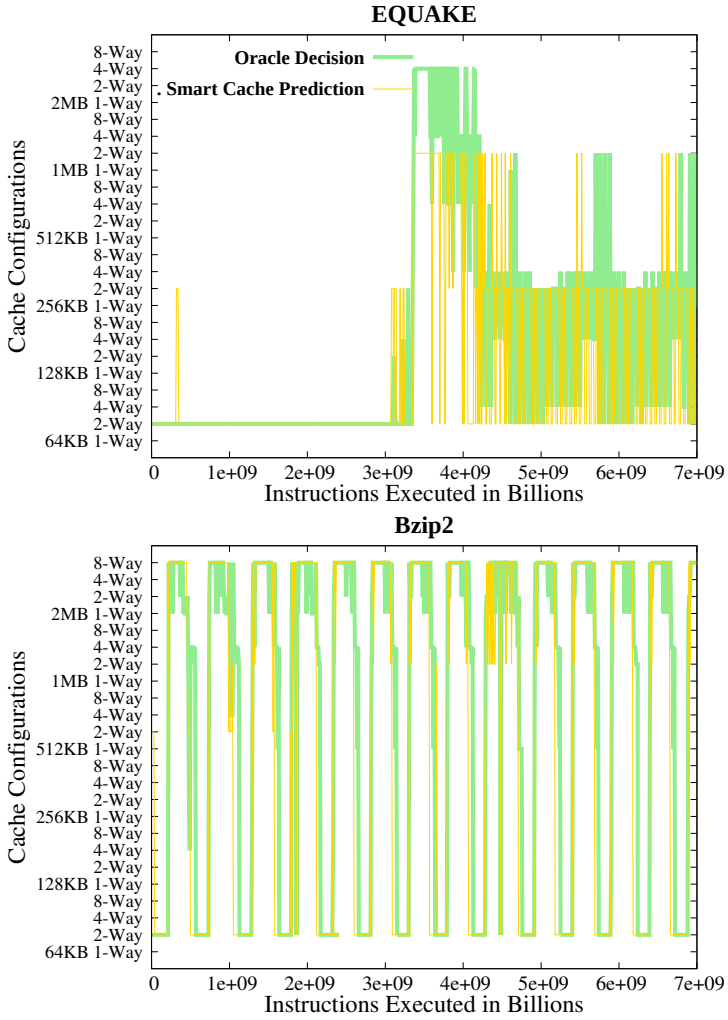


Figure 4.8: Dynamic cache configuration traces, illustrating the correspondence between the Oracle and the Smart cache reconfiguration behaviour of *equake* and *bzip2* applications. The y-axis shows different cache configurations and the x-axis shows the time interval of instructions executed.

over time with the oracle's selection.

4.5.1.3.1 Performance And ED These results show that the Smart cache is able to obtain significant energy-delay improvements with only minimal performance overheads. The average performance loss for smart cache approach is 1.8%, which is within the target value that was used to determine good cache configurations. Applications like *equake* and *galgel* incur 6.4% and 5.2% performance losses respectively which is primarily due to choosing smaller caches during the transition phase. In *crafty*, *facerec*, *mgrid*, *vortex* and *vpr*, the Smart cache performance is slightly over the actual limit of

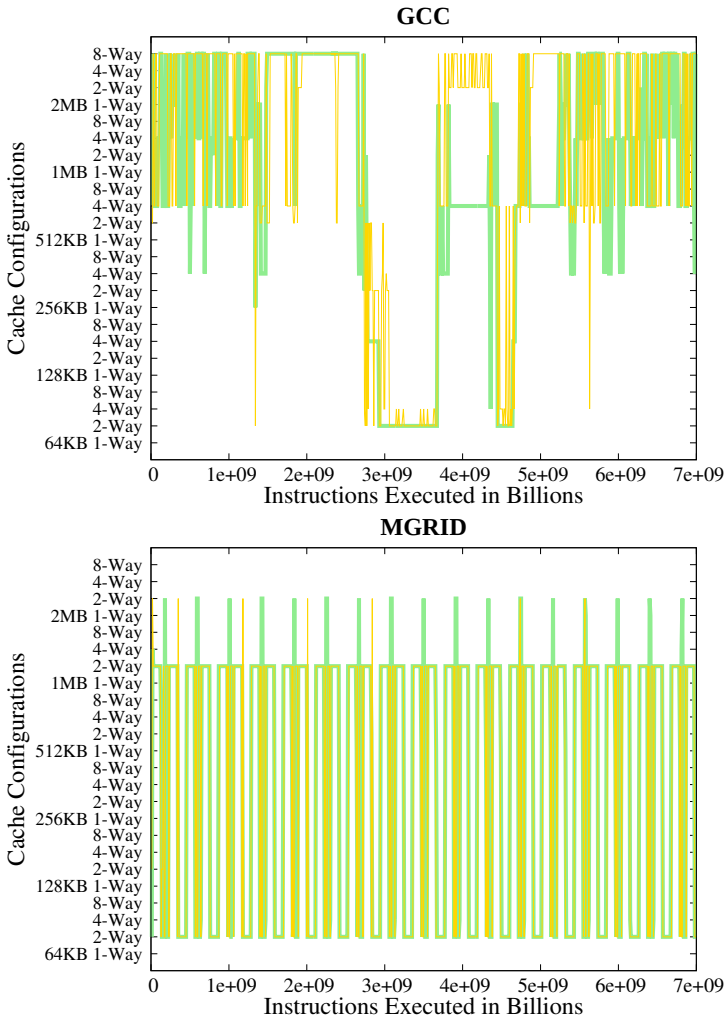


Figure 4.9: Dynamic cache configuration traces, illustrating the correspondence between the Oracle and the Smart cache reconfiguration behaviour of *gcc* and *mgrid* applications. The y-axis shows different cache configurations and the x-axis shows the time interval of instructions executed.

2%, whereas the oracle and best-static schemes are within the performance limit. The reason being that for a few small phases in the application, the smart cache model predicts a smaller cache size than required and hence it incurs extra performance losses, whilst also reducing energy-delay.

In terms of the energy-delay product, *eon* achieves 0.49 with no performance loss. This is mainly due to *eon* requiring a cache of 128KB. For *lucas*, the smart cache approach achieves a value of 0.83, whereas the oracle scheme is at 0.38. The cache statistics for *lucas* are similar to *parser* and *vpr*, when it is actually more similar to *applu*, *eon*, *gap* and *swim* in terms of cache size requirements.

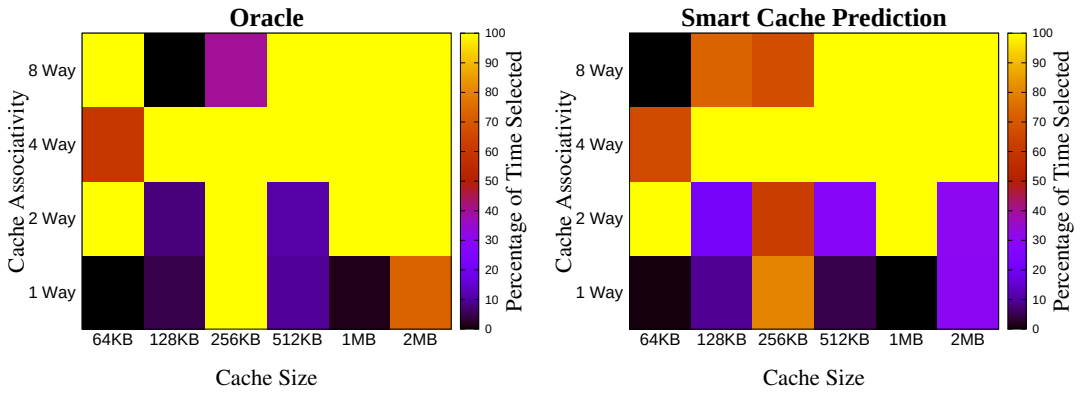


Figure 4.10: Heatmaps showing the distribution of level-2 configurations required by the oracle and Smart cache across all SPEC CPU 2000 applications.

In terms of the average energy-delay product, the smart cache achieves a value of 0.66 — a significant reduction compared to the best static approach. This shows the benefits of dynamic reconfiguration of the level-2 cache using smart cache approach.

4.5.1.3.2 Configurations Selected To consider how these savings are achieved, Figure 4.8 and Figure 4.9 show the predictions made by the Smart cache vary along the application. Also shown for comparison is the oracle approach. Due to space limitations, only the results from four representative benchmarks are shown.

In *bzip2*, there is a regular pattern of configurations that are required, alternating between a 64KB, 2-way cache and a 2MB configuration. It is clear from the diagram that smart cache approach accurately tracks the oracle and leads to the savings shown in figure 4.7.

The next two benchmarks (*equake* and *gcc*) have irregular patterns. For the majority of the time, the Smart cache can accurately determine the correct configuration to use. However, sometimes it predicts too small a cache size (in *equake*), leading to performance losses or too large a configuration (in *gcc*), leading to higher ED values than are optimal.

The final benchmark is *mgrid* which is interesting because smart cache obtains a lower ED value than the oracle. As can be seen in figure 4.8 and figure 4.9, this is due to the Smart cache accurately reconfiguring the cache as the oracle scheme does, but occasionally using a smaller size which leads to negligible performance losses but increased energy savings. Overall, figure 4.8 and figure 4.9 show that the Smart cache is able to track the configurations chosen by the unrealistic oracle scheme.

A summary of the configurations required by both the oracle and the Smart cache

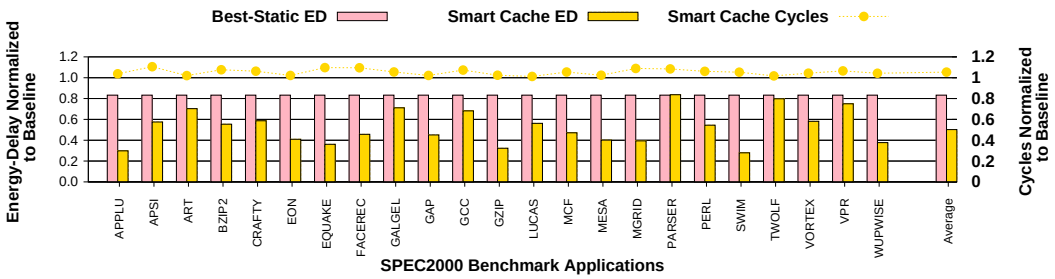


Figure 4.11: Combined performance and energy-delay characteristic of all three caches within the cache hierarchy, showing an overall reduction in energy-delay of 50%

can be seen in figure 4.10. The result is presented as a heat map, where darker blocks correspond to more frequently chosen configurations. These figures are averaged across all SPEC CPU 2000 applications.

The most frequently-used configuration is the 2MB, 4-way cache. In contrast, a direct-mapped cache is rarely chosen by either scheme, and nor is the smallest cache size of 64KB, apart from the 2-way configuration that is useful for certain benchmarks, as seen in figure 4.8 and figure 4.9. From these heat maps it is clear that the Smart cache's predictions are closely correlated to the configurations chosen by the oracle, providing further evidence of the accuracy of smart cache approach.

4.5.2 Cache Hierarchy Reconfiguration

Having shown the benefits of reconfiguring each cache individually, this section evaluates the effects of reconfiguring each cache in the hierarchy at the same time. Figure 4.11 shows the results. It shows the best static configuration and smart cache approach only. Results for the oracle scheme are not shown, because this would require a complete evaluation of the design space (i.e., 128,304 simulations) which is impractical within the current setup.

As figure 4.11 shows, applications such as *applu*, *art*, *eon*, *gap*, *gzip*, *lucas*, *mesa*, *twolf* and *wupwise* incur small performance losses of under 4%. However, other benchmarks experience larger losses, leading to an average performance loss of 5.3%.

On the other hand, there are significant improvements in the energy-delay values achieved. The smart cache approach is always better than the best static configuration with *swim* achieving a value of 0.27 and *applu* achieving 0.29. The reason behind the decrease in performance, when all caches change simultaneously, is the selection of inappropriate cache configurations during transition phases. This can be observed by

comparing figure 4.11 against other three individual cache changing schemes shown in figures 4.5 to 4.7.

For example, in *apsi*, individually changing the instruction and level-2 caches incurs less than 2% performance loss. However, when changing all caches at once, the smart cache model mistakenly selects too small a size for the data cache and this influences predictions made for the level-2 cache, increasing overall performance losses. A similar effect can be seen in *bzip2*, *equake*, *facerec*, *mgrid* and *parser*. Since the experiments are started from the beginning of each application and execute 60 billion instructions without using any profiled phase information, many small transition phases are encountered in the experiments. These transition phase boundaries could be easily identified by a phase detector [121, 103], which would allow us to vary the interval length and reconfigure more accurately. However, on average, the Smart cache approach achieves an energy-delay of 0.50, almost half that of the best static scheme.

4.5.3 Summary

This section has presented the results from the Smart cache approach. It is clear from figures 4.5 and 4.6 that reconfiguring the instruction and data caches does not bring many benefits. This is because of their small sizes in comparison to the level-2 cache, meaning that, relatively, they do not contribute as much energy to the total processor budget. However, as seen in figures 4.7 and 4.11, reconfiguring the level-2 cache can bring significant improvements in energy-delay. Therefore, dynamically reconfiguring the level-2 cache alone results in an overall cache hierarchy energy-delay reduction of 34% compared to a statically configured baseline cache.

4.6 Analysis of Smart Cache

Having shown the benefits of the Smart cache, the design space around it is considered. Specifically, a comparison is shown against existing state-of-the-art cache reconfiguration designs, then evaluate a linear regression model instead of a decision tree. Finally, this section shows that this approach is robust across benchmark suites by training on SPEC CPU 2000 and running on SPEC CPU 2006.

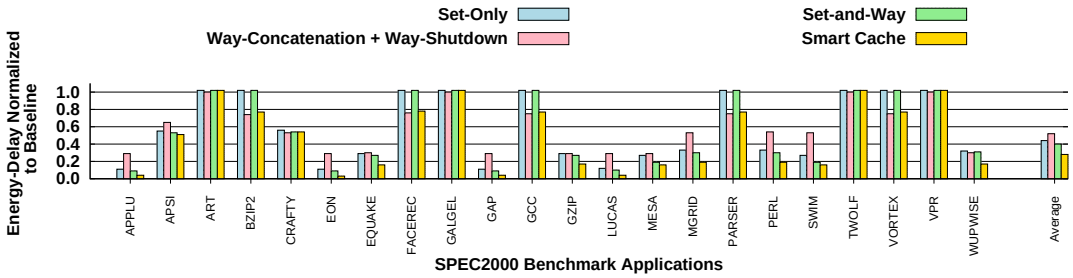


Figure 4.12: Energy-delay values for different cache architectures running on the baseline level-2 cache configuration.

4.6.1 Comparison With Prior Work

The first evaluation compares the smart cache architecture with prior state-of-the-art designs. Figure 4.12 shows set-only cache [105], which can increase/decrease cache size; Way Concatenation cache [143], which concatenates one or more ways to get the desired associativity and also uses way-shutdown to turn-off the unused ways to reduce leakage power. Set and Way cache [138], which incorporate both [8, 105] schemes and the Smart Cache. The energy-delay product achieved when running each benchmark on the best static configurations for that application for each cache architecture is shown. The best static configuration is the one that has the lowest energy-delay and a maximum 2% performance loss, from all the possible configurations.

For benchmarks such as *art*, *bzip2*, *facerec*, *galgel*, *gcc*, *parser*, *twolf*, *vortex* and *vpr* the best static configuration is 2MB with eight-way associativity so there are no energy savings achievable for any cache architecture. The set-only, set-and-way and Smart approaches consume around 1.7% more energy compared to way-concatenation, because the latter does not use extra tag-bits that other architectures require to change the cache size. For some benchmarks, like *bzip2*, *facerec*, *gcc*, *parser* and *vortex*, the set-only and set-and-way approaches do not do well compared to the way-concatenation and Smart caches. The reason for this is that these benchmarks require a 2MB cache with two-way associativity which is only offered by way-concatenation and Smart cache. For these architectures, dynamic energy is reduced by accessing fewer ways, which is not possible in the set-only and set-and-way caches.

For others benchmarks, such as *applu*, *eon*, *gap* and *lucas* significant energy-delay reductions can be achieved. This is due to smart cache approach accessing fewer sets and ways as compared to the set-only and set-and-way approaches for lower associativity. It can also be seen that no single approach can provide good energy-delay values

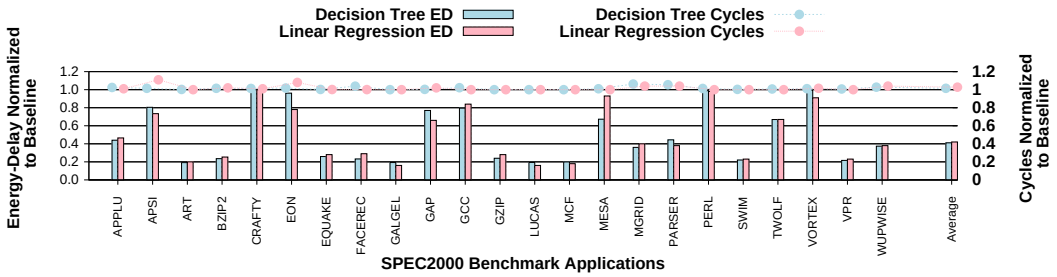


Figure 4.13: Performance and energy-delay comparison of decision tree and linear regression models when used to reconfigure the instruction cache.

for all applications.

Overall, the average level-2 cache energy-delay achieved by the smart cache approach is 0.28, which is 18% better than set-only and set-and-way approaches and 25% better than the way-concatenation with way-shutdown approach. This clearly demonstrates the benefits of using smart cache architecture for cache reconfiguration. The next section now harnesses this flexibility to dynamically reconfigure level-2 cache to obtain further power savings.

4.6.2 Impact of Predictive Model

It is interesting to see how the Smart cache would behave, when a different machine learning model is selected to guide its reconfiguration decisions. Now, considering a linear regressor as an alternative model and results for different caches are shown in figures 4.13 to 4.15.

4.6.2.1 Instruction Cache

Figure 4.13 shows the performance and energy-delay of the Smart cache when using a decision tree and linear regression model and reconfiguring the instruction cache alone. On an average the performance and energy-delay of the two models are almost same, leading to a reduction of 59% in energy-delay compared to the baseline. However in applications like *apsi* and *eon*, the decision tree model's prediction is within the chosen performance limit of 2%, whereas the linear regression model incurs 11% and 8% for *apsi* and *eon* respectively. In *mesa* the linear regression model predicts a bigger cache size of 32KB than the required 16KB, which is accurately predicted by the decision tree model.

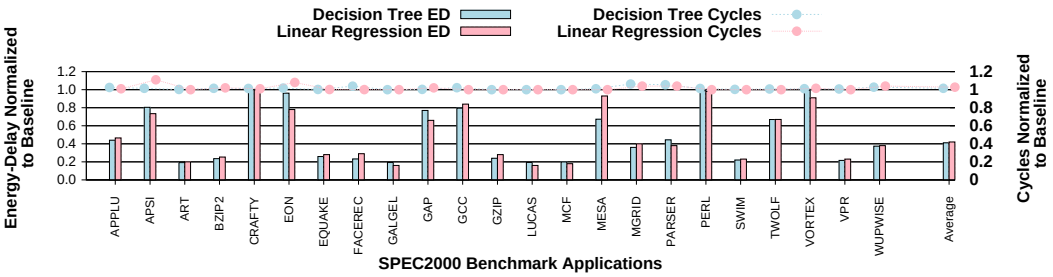


Figure 4.14: Performance and energy-delay comparison of decision tree and linear regression models when used to reconfigure the data cache.

4.6.2.2 Data Cache

Now coming to the data cache, shown in figure 4.14, it can be seen that on an average the performance and energy-delay of the two models are almost same. Thus, both models offer a reduction of 66% in energy-delay compared to the baseline. However, in applications like *crafty* and *galgel* the linear regression model predicts the wrong cache configuration. This leads to a performance degradation of 6% in *crafty* and an energy-delay of 0.40 above the decision tree model.

4.6.2.3 Level-2 Cache

Now considering the last level cache. Figure 4.15 shows the prediction results of dynamic reconfiguration using the decision tree and linear regression models. Once more, on average both models have similar performance and offer a significant energy-delay reduction of 64% compared to the baseline configuration.

In applications like *apsi*, *facerec*, *galgel* and *parser* the linear regression model incurs 9%, 10%, 8% and 8% performance loss respectively compared to the decision tree model which incurs less than 4% performance loss in all the above applications. On an average both the decision tree and linear regression models perform similarly in terms of both performance and energy-delay. However, a decision tree performs better on a per-application basis and is more simple to implement in hardware compared to the linear regression model. Hence, the choice of a decision tree model to guide smart cache reconfiguration was justified.

4.6.3 Reconfiguring SPEC CPU 2006

Although leave-one-out cross-validation is employed to evaluate the decision tree model, it is interesting to consider how this approach would perform by training on one bench-

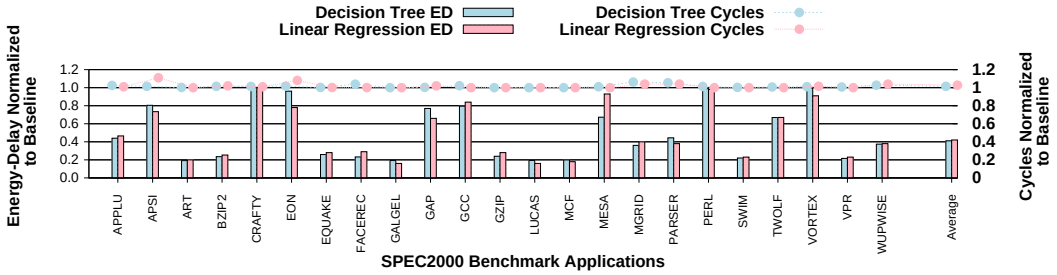


Figure 4.15: Performance and energy-delay comparison of decision tree and linear regression models when used to reconfigure the level-2 cache.

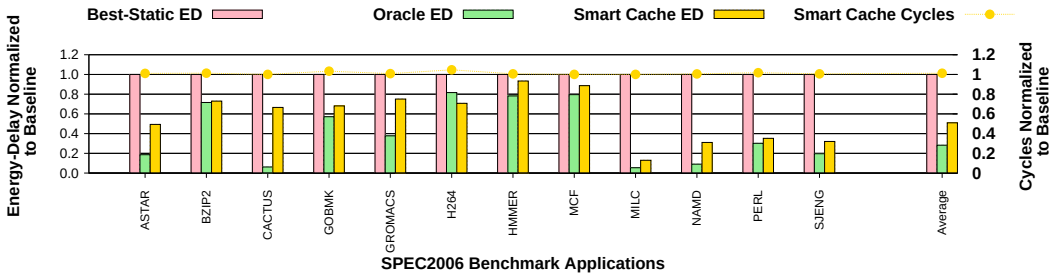


Figure 4.16: Performance and energy-delay characteristic of the unified level-2 cache for SPEC CPU 2006 after training the decision tree model on SPEC CPU 2000.

mark suite and reconfigure the caches for another takes place. Figure 4.16 shows the results from training on SPEC CPU 2000 benchmarks and then dynamically reconfiguring the level-2 cache for SPEC CPU 2006. Only a subset of the benchmarks are shown, because several of them could not be compiled in the Alpha environment.

What is immediately obvious is that the decision tree model can accurately capture the benefits available on a different benchmark suite. The average performance loss is only 1.2% and the average energy-delay is 0.51. This comes from a dramatic energy-delay value of 0.13 for *milc*, 0.31 for *namd*, 0.35 for *perl* and 0.32 for *sjeng*. The results here show that the decision tree model can be applied across benchmark suites to achieve the benefits of dynamic cache reconfiguration with minimal performance loss.

4.7 Multicore Systems

Now final evaluation considers the impact of sharing within the last level cache and the benefits achievable by the Smart cache architecture in this scenario.

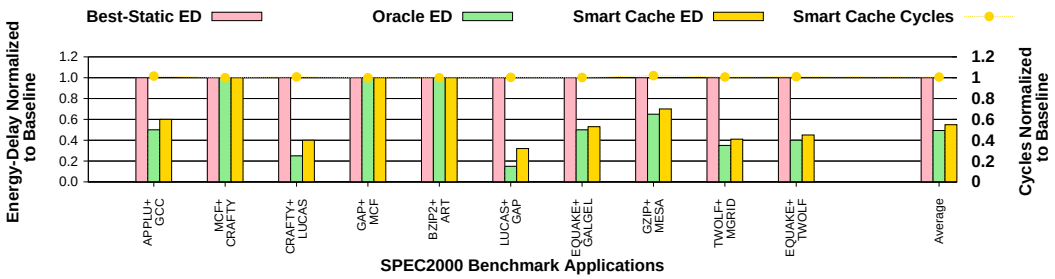


Figure 4.17: Performance and energy-delay characteristic of two-core workloads.

4.7.1 Simulation Infrastructure

Compared to all other experiments in this chapter, a multiprogrammed version of HotLeakage is used, with the same updates as in the single core experiments. This includes the addition of separate Re Order Buffer (ROB) entries, Load Store Queue (LSQ), instruction and data cache for each core. They all share the Level-2 cache. Due to the large simulation times, 20 billion instructions are fast-forwarded from each benchmark, warmed-up the caches and branch target buffers for 500 million cycles, reset the statistics and then ran for 1 billion cycles per application.

The level-2 cache is altered to a 4MB, 8-way configuration for the two-core workloads and a 8MB, 16-way cache for the four-core workloads, to account for the increased requirements of the benchmarks. The workloads are randomly selected from those applications that would run correctly in this multiprogrammed simulator. Again the leave-one-out cross-validation is used to train the Smart cache.

4.7.2 Two Cores

The results for two-core system are shown in figure 4.17. On an average the Smart cache offers an energy-delay of .55 when compared to the baseline and best-static configurations. For applications like *crafty+lucas* and *lucas+gap*, smart cache scheme predicts bigger cache configurations (1.5MB) than the required 1MB. Nevertheless, on an average, the Smart cache energy-delay is just 5% away from the oracle value, incurring less than 1.5% performance loss compared to the baseline. There is no energy saving when running *mcf+crafty*, *gap+mcf* and *bzip2+art* workloads, as these applications require the entire cache. In these situations, the Smart cache recognizes the need for a large cache and does not attempt reconfiguration. However in *lucas+gap* a significant reduction in energy-delay of 70% is achieved by the Smart cache.

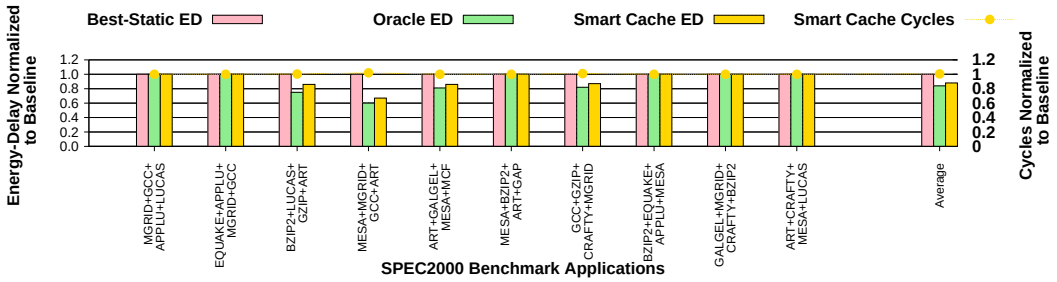


Figure 4.18: Performance and energy-delay characteristic of four-core workloads.

4.7.3 Four Cores

Finally, results on a four-core system are shown in figure 4.18. Here the Smart cache offers an energy-delay reduction of 12% on average. For workloads like *bzip2 + lucas+ gzip + art*, *mesa+mgrid+gcc+art*, *art+galgel+mesa+mcf* and *gcc+gzip+crafty+mgrid* smart cache achieves energy-delay reductions of 14%, 33%, 15% and 12% respectively. These results clearly show that the Smart cache can be used in both single and multicore scenarios where caches are shared among several different applications.

4.8 Summary

This chapter has presented a novel configurable cache architecture and a decision tree machine learning model that dynamically predicts the best cache configuration for any application. The main goal is to reduce both dynamic and static energy without losing performance. This chapter has demonstrated that the smart cache approach reduces energy-delay by 0.17 in the data cache and 0.34 in the level-2 cache with less than 2% performance degradation in comparison to the baseline cache. Furthermore, this has demonstrated that smart cache approach generalizes by predicting one benchmark suite after training on another, and applying the Smart cache to multiprogrammed scenarios where the last-level cache is shared among several different applications. The next chapter will discuss the shared LLC cache architecture present in CMPs and proposes an energy efficient cache partitioning scheme for the same.

Chapter 5

Cooperative Partitioning

The demand for high performance computing systems requires processor vendors to increase the number of cores per Chip Multiprocessor (CMP). As the processor count per chip increases, effective energy management of the Last Level Cache (LLC), which is the largest cache and common to all cores, becomes increasingly important. This energy constraint is enforced by the fixed Thermal Design Power. Most of the previous work on cache energy savings are based on single-core designs and are either focused on turning off parts of the cache memory to save static energy or on predicting the way that will be accessed, thereby saving dynamic energy. These techniques are mostly inapplicable to the LLC because of differing problem constraints.

The solution proposed in Chapter 4 incurs re-mapping of address, when cache size is increased by varying the cache sets. This chapter proposes a cache partitioning that depends on varying only the cache ways. This avoids re-mapping of addresses when the cache size is increased by varying the cache sets as proposed in Chapter 4. The solution proposed in Chapter 4 still holds good for Level 1 caches.

For Level 2 caches, this chapter proposes an LLC energy-saving scheme, specific to shared memory multi-core CMPs with multi-programmed workloads that reduces both dynamic and static energy with minimal performance degradation. The proposed partitioning scheme is evaluated on two-core and four-core systems, showing that it obtains an average dynamic and static energy savings of 35% and 25% respectively, compared to a fixed partitioning scheme. In addition, Cooperative Partitioning maintains high performance while transferring ways five times faster than an existing state-of-the-art technique.

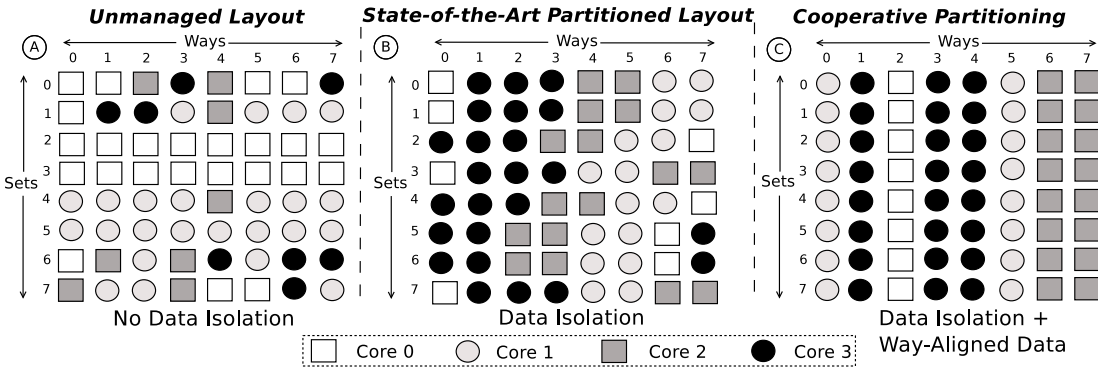


Figure 5.1: Data allocation across partitioning schemes in shared last-level caches.

5.1 Introduction

On-chip caches play a significant role in improving the performance of a processor. Within a chip multiprocessor (CMP), a multi-level cache hierarchy is employed, with the last-level cache (LLC) being the largest and often shared among all cores on the chip. Decreasing its energy consumption is important because it is responsible for a significant fraction of the total processor power budget. However, any efficient energy-saving technique should cause minimal or no performance degradation.

Cache energy reduction techniques have been widely studied in the past. Most work has focused on single-core designs. These turn off parts of the cache, to reduce static energy, or predict the ways that will be accessed, to reduce dynamic energy. However, these schemes are not directly applicable to a CMP LLC due to the filtering effects of the higher cache levels making access patterns hard to predict.

In contrast, there has been significant recent work in partitioning a shared LLC for performance [125, 124, 137, 136]. Applications are restricted to a number of logical ways within the cache, giving the most resources to the programs that obtain the most benefit from them. While these techniques can unlock significant performance increases, they do not consider energy saving when partitioning.

This chapter takes a novel approach to LLC partitioning by forcing data belonging to each core to be *way-aligned* across all sets. Figure 6.2 gives an example of how cooperative partitioning approach differs from existing partitioning schemes. An unmanaged cache is shown in A, where data belonging to the cores is entirely mixed across sets and ways. In B, a cache partitioning technique has been applied so that the number of ways owned by each core is constant across all sets. However, within the sets, data from each core can reside in any way. The proposed scheme is shown in C. It applies the same partitions as in B, but enforces data way-alignment so that a way is

owned entirely by a single core at a time.

The energy savings achieved are two-fold. First, dynamic energy can be reduced on each access because a core only needs to consult the ways that it currently owns. The scheme guarantees that its data will never be found anywhere else in the cache. Second, when a whole way is unused by any core, it can be turned off to save static energy. Implementing the technique in a partitioned architecture combines large energy savings with high performance.

The proposed scheme is named *Cooperative Partitioning*, because cores cooperate with each other after partitioning, to migrate ways between themselves. The state-of-the-art cache partitioning along with the fixed partitioning is implemented in the simulation infrastructure described in Section 5.3. Using cooperative partitioning in a two-core system brings dynamic energy savings of 32% and static energy savings of 25% compared to a fixed partitioning scheme. In a four-core environment, dynamic and static energy savings of 31% and 20% respectively are achieved. In addition, due to the cooperative takeover algorithm for transferring ways, migration is five times faster, on an average, than a state-of-the-art partitioning scheme and flushes less data back to memory.

The rest of this chapter is structured as follows. Section 5.2 describes the cache monitoring scheme and partitioning algorithm and explains the internals of the cache architecture. This section also discusses the overheads associated with the cache reconfiguration. Section 5.3 describes the experimental methodology, workloads, and metrics used for evaluation. Section 5.4 evaluates the proposed approach on two-core and four-core systems and Section 5.5 analyses the reasons behind the results. Finally, Section 5.6 concludes.

5.2 Cooperative Partitioning

The cooperative partitioning scheme is split into two distinct parts. The first monitors cache usage and determines the optimal partitions for the running applications. The second enforces the required partitions, enabling static and dynamic energy savings. As is common in last level caches, this section assumes accesses are serial. Therefore dynamic energy savings come from the tag side only.

An overview of the cache partitioning system is shown in Figure 6.3. During the first phase, LLC accesses are monitored and partitioning decisions are made periodically, about the number of ways to allocate to each core according to their cache re-

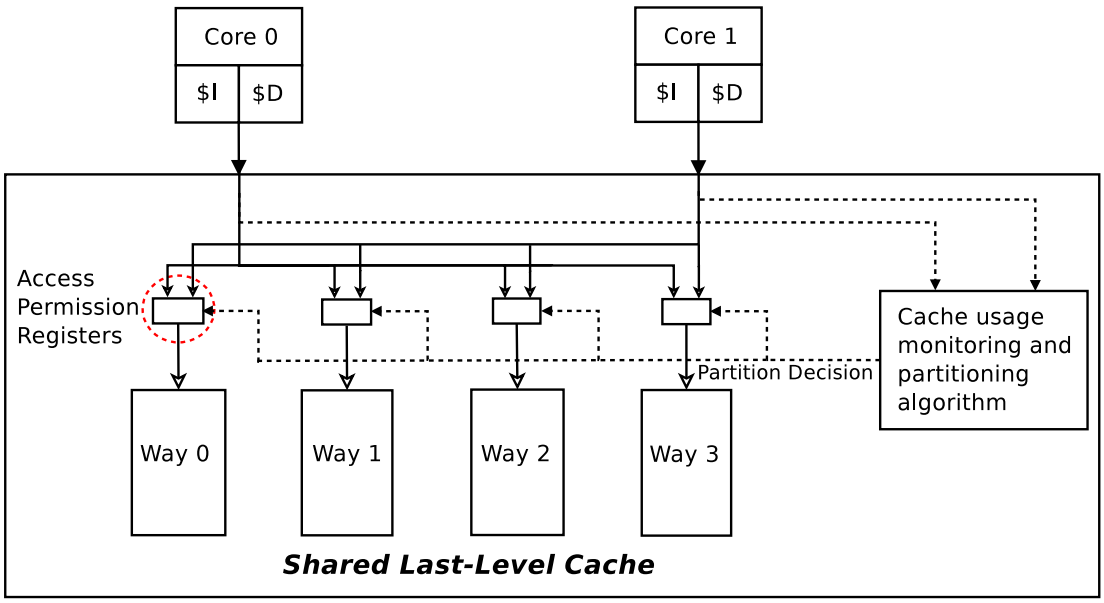


Figure 5.2: An overview of the cooperative partitioning architecture.

quirements. In the second phase, this information is used to set the appropriate access permission registers that determine how each core can access each LLC way. During this phase, ways are gradually migrated between cores or turned off.

To enforce the required partitioning decisions and enable way-alignment of data, two new registers for each way called the read access permission (RAP) and write access permission (WAP) registers are introduced. These allow specific cores to read and write in each way.

First, it describes the cache monitoring scheme, then explain the RAP and WAP registers in more detail. Then it describes how reconfiguration is achieved by transferring ways between cores. This is called as cooperative takeover and an example of its workings is given. Finally, this section describes the overheads associated with the cache architecture.

5.2.1 Usage Monitoring and Partitioning

The cache architecture builds on prior work to determine the optimal partitions for the LLC. As in state-of-the-art schemes, the proposed scheme targets a cache that is shared among multiprogrammed workloads [14, 19, 75, 109, 136]. Accesses are tracked by utility monitors [109] for computing each application's use of the cache. Other partitioning schemes have also made use of these monitors [136], although they could also be specified through the operating system [112].

Algorithm 1: Cores obtain extra ways when their performance increases above a threshold.

```
balance = N; /* Number of Blocks to be allocated */
allocations[i] = 0; /* For each competing application, i */
prev_max_mu = 0;
while balance do
    foreach application i do
        alloc = allocations[i];
        max_mu[i] = get_max_mu(i, alloc, balance);
        blocks_req[i] = min blocks to get max_mu[i] for
        i;
    winner = application with maximum value of
    max_mu;
    /***** Modified implementation starts here *****/
    This following are the modifications that are added
    to UCP [109].
    if |prev_max_mu - max_mu| < (prev_max_mu *
    T_hold) then
        allocations[winner] += blocks_req[winner];
    balance -= blocks_req[winner];
    prev_max_mu = max_mu;
    /***** End of modifications *****/
return allocations;
```

```
get_max_mu(p, alloc, balance):
max_mu = 0;
for j=1; j<=balance; j++ do
    mu = get_mu_value(p, alloc, al-
    loc+j);
    if mu > max_mu then
        max_mu = mu;
return max_mu;

get_mu_value(p, a, b):
return = (miss_a - miss_b)/(b-a);
```

A modified Utility based Cache Partitioning (UCP) look-ahead algorithm [109] is used to determine partitions, shown in Algorithm 3. This contains a threshold value that is used when allocating ways to a core. The threshold controls the decrease in miss-ratio for each application, preventing each core from being awarded additional ways unless it can significantly benefit from them. Therefore, after running the algorithm, there may be ways that are not allocated to any core. These can be turned off for static energy savings with minimal loss of performance.

5.2.2 Cache Partitioning Control

To control each core's access to the ways, and enforce way-aligned data, an additional RAP register and WAP register is introduced for every way within the cache. Each RAP register has one bit per core to indicate whether that core can read from the associated way. This is used in conjunction with the WAP register for that way. This

Algorithm 2: Setting the RAP and WAP registers to initiate cooperative takeover.

```
Pre = Previous way allocations per core;
Cur = Current way allocations per core;
for i = 0; i < n; i = i + 1 do
    if Pre[i] < Cur[i] then /* Core i acts as a recipient */
        receive[i] = Cur[i] − Pre[i]; donate[i] = 0;
    else if Pre[i] > Cur[i] then /* Core i acts as a donor */
        donate[i] = Pre[i] − Cur[i]; receive[i] = 0;

for i = 0; i < n; i = i + 1 do
    for j = 0; j < n; j = j + 1 do
        if receive[i] > 0 and donate[j] > 0 then
            if donate[j] > receive[i] then
                donation = receive[i];
            else
                donation = donate[j];
            for d = 0; d < donation; d = d + 1 do
                w = Random way owned by core j;
                RAP[w][i] = 1; WAP[w][i] = 1;
                WAP[w][j] = 0;
                receive[i] -= 1; donate[j] -= 1;

/* Turn ways on or off */
for i = 0; i < n; i = i + 1 do
    if donate[i] > 0 then
        for d = 0; d < donate[i]; d = d + 1 do
            w = Random way owned by core i;
            WAP[w][i] = 0;
        donate[i] = 0;
    else if receive[i] > 0 then
        for r = 0; r < receive[i]; r = r + 1 do
            w = Random way currently off;
            RAP[w][i] = 1; WAP[w][i] = 1;
        receive[i] = 0;
```

also has one bit per core and indicates whether the core has permission to write to the way or not.

For each core and way, there are three possible modes of operation. If both registers are set for a particular core, then this core can both read and write in that particular way. Otherwise, if the RAP register is set and the WAP register is unset, then the core has only read permission for accessing that way. If both registers are unset, then the core can neither read nor write that way.

Only one core can have full access (RAP set and WAP set) to a particular way at any given time. In fact, under normal conditions only one core can have any access to the way. However, during a transition period, when reconfiguration is taking place, one core can have full access and another can have read-only access. This lasts until the whole way has been transferred from one core to the other and is discussed in more detail in Section 6.2.3. Algorithm 4 describes how the RAP and WAP registers are set at the beginning of a transition period.

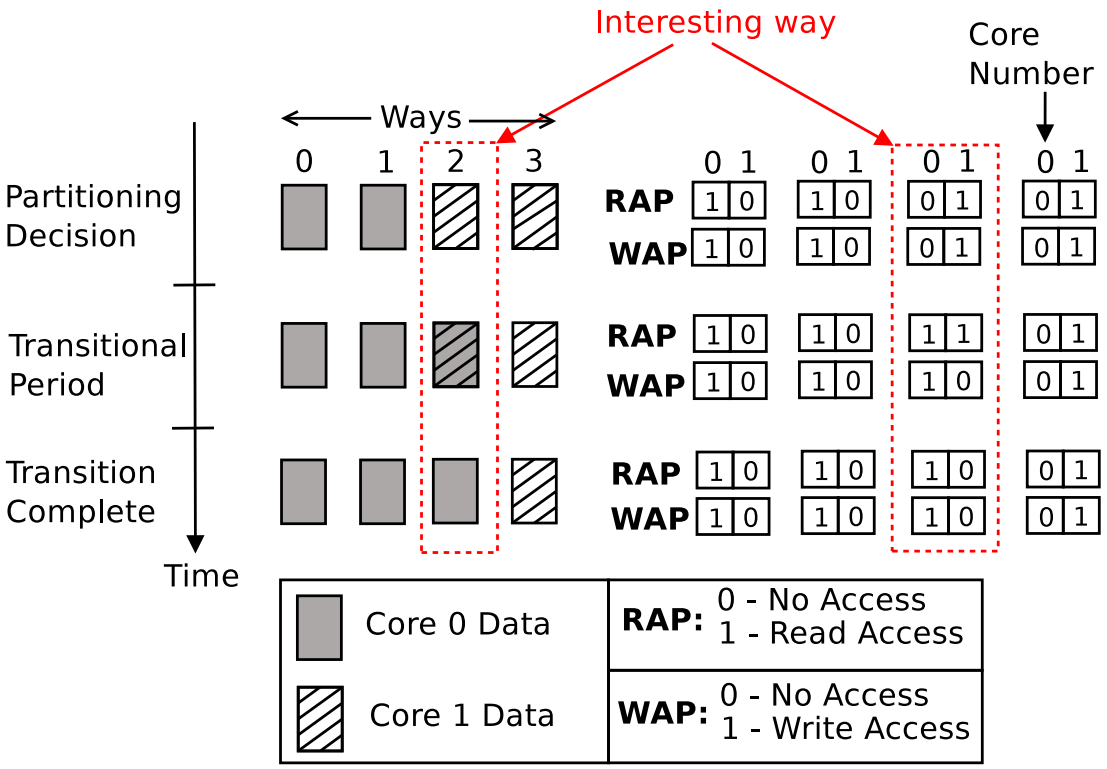


Figure 5.3: RAP and WAP register changes when transferring way 2 between cores.

The RAP and WAP registers serve three purposes. First, they enforce the cache partitioning that is currently in operation by restricting cores' accesses to only the ways that they are allocated. Second, they enable dynamic energy savings because cores only need to access the ways that they have permission for, rather than all ways within the cache. Third, when no cores have access to a particular way (RAP and WAP unset for all cores), then the whole way can be turned off for static energy savings.

Figure 5.3 shows an example of the RAP and WAP registers before, during and after a transition period. Initially both cores own two ways and the registers are set accordingly. A partitioning decision is then made that transfers way 2 to core 0. To allow this, core 0 gets read and write access to way 2, and core 1's write permission is revoked. After the transition period, core 0 has full control of the way and core 1's read permission is also withdrawn.

5.2.3 Cache Reconfiguration

Once the RAP and WAP registers have been set, the cache must be reconfigured to the new partitioning that is required. To achieve this, a new technique called cooperative takeover is introduced. In this scheme, for each way to be transferred, the donor and

recipient cores cooperate to quickly flush dirty data back to memory and allow the recipient core to take full ownership of all lines in the way.

The proposed scheme avoids the cost of immediately flushing data back to memory, but quickly transfers ownership of the whole way, enabling fast realisation of the dynamic energy savings that can be achieved (when the donor core no longer accesses this way). During transitional periods, dynamic energy consumption is higher than normal because multiple cores access the ways that are being transferred. Therefore, it transfer ways as quickly as possible to minimise the length of time that the way is transitioning.

To enable cooperative takeover, the cache is augmented with a takeover bit vector for each core that is the size of the number of sets in the cache (i.e., one bit per set per core). The donor cores' bit vectors are reset at the start of a transition period. Whenever a donor core accesses a particular set, dirty data is flushed back to main memory from the ways that it is transferring. This happens whether it hits or misses on that particular access. At the same time, the bit for that particular cache set in the core's bit vector is set. Additionally, whenever a recipient core accesses a particular set, dirty data is flushed back to memory from the ways that it will be receiving. Again, this occurs whether it hits or misses on that access. In this situation, the bit for that cache set in the donor core's bit vector is set.

The donor core knows that it is donating ways because it has read permission on those ways, but not write permission. The recipient core knows that it will receive certain ways because it will have read and write permissions to the ways, but another core will also have read permission. When donating or receiving ways, dirty data is flushed in all ways with read permission on each access.

Bit vectors are reset at the start of a transition period for each donor core that is giving away a way. This could interfere with a prior transition of a different way from a donor core that is still in progress. In this situation, the bit vector is still reset and the only result is that the first transition will take longer to complete. However, this situation is rare and is not seen it in any of the experiments.

5.2.4 Cooperative Takeover Example

Figure 6.4 shows an example of cooperative takeover in practice. In this example there are two cores and four cache ways. Initially, the partitioning decision has just been made and two ways are assigned to each core, but core 1 will donate way 2 to core 0.

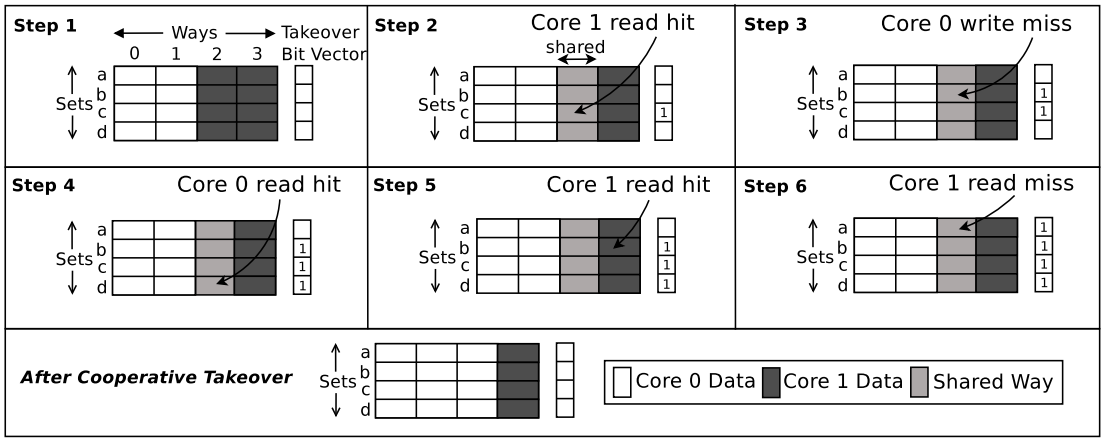


Figure 5.4: An example of cooperative takeover where core 1 will donate a way to core 0. Whenever either core accesses a set, dirty data is flushed back to memory. Once all sets have been accessed by at least one core, the way can be owned entirely by core 0.

The takeover bit vector for core 1 is totally unset. In the second step, core 1 performs a read that hits in set c in the cache. Its own dirty data from this set in way 2 is flushed back to memory and the takeover bit is set.

Following this, core 0 writes to the cache which misses in set b. In this case, core 1's dirty line from set b, way 2 is flushed and the corresponding takeover bit set as before. When the new line comes in from memory, it can be placed in way 2 instead of replacing an existing line in another way.

Core 0 then has a read hit in set d. In this case the line in way 2 is not dirty so does not need flushing, but the takeover bit is still set. In the fifth step, core 1 has a read hit in set b. However, the line in way 2 is now owned by core 0 and, even though it is dirty, does not need flushing back to memory. Core 1 can see this because the corresponding takeover bit is already set. Finally, core 1 has a read miss in set a. Again, no dirty data needs flushing, and the takeover bit is set. However, when the line comes into the cache, it will replace the data in way 3 (core 1's only way).

At this point, all the takeover bits are set and therefore core 0 takes complete ownership of way 2. This is achieved simply by resetting the bit for core 1 in the RAP register for way 2, meaning that it no longer has read permission for that way (write permission had already been withdrawn).

5.2.5 Reconfiguration Overheads

There are four types of overhead associated with the cache partitioning and reconfiguration scheme. These are the changes to the replacement policy, hardware overheads of implementation, and the performance and power overheads of carrying out the partitioning.

5.2.5.1 Replacement Policy

The scheme proposed in [109] is used in cooperative partitioning, where an extra two bits are added to each tag entry to distinguish data belonging to each core. Algorithm 4 determines which ways will be transferred between cores, then the replacement algorithm links the corresponding ways accordingly [23, 63, 124].

5.2.5.2 Hardware Overheads

As in other schemes [136], the proposed scheme uses an existing cache monitoring scheme to track the usage of each set by each core and requires the same hardware as this [109]. To implement cooperative takeover, it only requires one bit vector for each core for each set, along with RAP and WAP registers for each way. In the 4MB L2 cache, used for four-core experiments in Section 5.4, this comes to a little over 8k bits. Table 6.1 details the requirements for the two caches that have been taken to study.

5.2.5.3 Performance Overheads

When transferring a way from one core to another, blocks from each set are selected to be given to the recipient core. State-of-the-art schemes are free to choose any block within each set; selecting the LRU block is one method [136]. Cooperative partitioning keeps the data way-aligned, so does not have the flexibility to choose blocks on a per-set basis. This makes the scheme closer in performance to a random choice of replacement block. However, in practice, this causes a negligible performance loss compared with prior work and is more than offset by the energy savings.

5.2.5.4 Power Overheads

Since the cache has extra circuitry for monitoring and partitioning, it consumes more power than a regular cache. However, it can realise considerable savings in dynamic and static energy, which far outweigh the overheads incurred. Nevertheless, all power overheads are included in the simulated results in Section 5.4.

	Two Core		Four Core	
Hardware Description	Details	Bits	Details	Bits
Takeover Bit Vectors	2048 * 2	4096	2048 * 4	8192
RAP	8 * 2	16	16 * 4	64
WAP	8 * 2	16	16 * 4	64
Total		4128		8320

Table 5.1: Summary of the hardware overheads of the proposed scheme for two-core and four-core systems.

5.2.6 Summary

This section has described Cooperative Partitioning as a high-performance, energy efficient cache partitioning scheme. This chapter introduces RAP and WAP registers to control access to cache ways, keeping data way-aligned and enabling unused ways to be turned off. During transition periods, when ways are being transferred between cores, both donor and recipient cores cooperate to flush dirty data back to memory. This enables the recipient to quickly take ownership of the ways and maximises the time when dynamic energy savings can be realised.

5.3 Experimental Methodology

This section describes the environment used to evaluate the proposed cache architecture.

5.3.1 Simulator

The proposed cooperative partitioning is implemented in Marss-x86 [102]. Table 6.2 shows the configuration of the system. A 4-wide, x86-based out-of-order processor with a 7 stage pipeline is simulated. A multi-core system with a two-core and a four-core is modelled to fully evaluate the effects of sharing and partitioning the last level cache. All level 1 caches are private and all processors share a common level 2 cache. The DRAM conflicts and bus queueing delays are modelled and Cacti [131] at 45nm is used to get energy information. Finally, a 5 million cycle phase interval for monitoring and partitioning decisions is assumed, as in prior work [109].

Parameters	Configuration
Processor	4-wide, out-of-order, 7 stage pipeline
ROB	128 entry
LSQ	48 entry
Branch Pred.	Gshare, minimum 10 cycle misprediction penalty
BTB	1024 entry, 4-way set-associative
L1 ICache	32kB, 64B lines, 4-way, 2 cycle lat
L1 DCache	32kB, 64B lines, 4-way, 2 cycle lat
Shared L2	2MB, 64B lines, 8-way, 15 cycle lat (two-core)
	4MB, 64B lines, 16-way, 20 cycle lat (four-core)
MSHR	128 entry
Memory	8 DRAM banks, 400 cycle lat, 64 outstanding reqs

Table 5.2: System configuration.

5.3.2 Workloads

All C and C++ benchmarks from SPEC CPU2006 [122] are taken, which totals 19 applications; FORTRAN benchmarks could not be incorporated into the simulation environment. To select groups to run in parallel, benchmarks are arranged into categories according to their misses per kilo instructions (MPKI) within the last level cache. Table 5.3 shows this classification.

A group of 14 two-application workloads are created by randomly selecting benchmarks so that there was at least one highly memory intensive program (MPKI > 5) in each group. The 14 four-application workloads were created by randomly selecting applications so that groups contained at least one highly memory intensive and one mediumly memory intensive program ($1 <$ MPKI < 5). These are shown in table 5.4.

All benchmarks are run using the reference inputs, after first skipping the initialisation routines that is discovered through source code inspection. Having fast-forwarded through initialisation, the caches and branch predictor are warmed for 5 million cycles. Then, at least 1 billion instructions per application are simulated, as is common practice [44, 136]. Statistics are reported for 1 billion instructions per benchmark, but all applications continued running until the last program in the group had reached 1 billion instructions, to keep contending for cache resources.

Group	Benchmark	MPKI	Group	Benchmark	MPKI
High	Gobmk	9	Low	DealII	0.8
	Lbm	20.1		Gromacs	0.32
	Sjeng	9.5		H264ref	0.89
	Soplex	18		Milc	0.96
Medium	Astar	4.8		Namd	0.25
	Bzip2	3.2		Omnetpp	0.26
	Calculix	1.1		Perlbench	0.98
	Gcc	4.92		Povray	0.1
	Libquantum	3.4		Xalan	0.6
	Mcf	4.8			

Table 5.3: Workload classification based on misses per kilo instructions (MPKI). The High group has MPKI $>$ 5, Medium is 1 $<$ MPKI $<$ 5 and Small has MPKI $<$ 1.

Two Core Workloads		Four Core Workloads	
G2-1	Soplex, Namd	G4-1	Gobmk, Gcc, Perl., Xalan
G2-2	Soplex, Milc	G4-2	Sjeng, Lbm, Calculix, Om.
G2-3	Gobmk, H264.	G4-3	DealII, Sjeng, Soplex, Namd
G2-4	Lbm, Povray	G4-4	Soplex, Sjeng, H264., Astar
G2-5	Gobmk, Perl.	G4-5	Lbm, Libq., Gromacs, Mcf
G2-6	Lbm, Bzip2	G4-6	Gobmk, Libq., Namd, Perl.
G2-7	Lbm, Astar	G4-7	Lbm, Sjeng, Povray, Om.
G2-8	Lbm, Soplex	G4-8	Lbm, Soplex, H264., DealII
G2-9	Soplex, DealII	G4-9	Lbm, Xalan, Milc, Soplex
G2-10	Sjeng, Calculix	G4-10	Sjeng, Povray, Milc, Gobmk
G2-11	Sjeng, Xalan	G4-11	Gobmk, Libq., H264., Gromacs
G2-12	Soplex, Gcc	G4-12	Soplex, Astar, Om., Milc
G2-13	Sjeng, Povray	G4-13	Soplex, Gcc, Libq., Xalan
G2-14	Gobmk, Om.	G4-14	Soplex, Bzip2, Astar, Milc

Table 5.4: Workload groupings.

5.3.3 Evaluation Metrics

To measure system performance a weighted speedup metric is used. This shows the reduction in execution time for each benchmark compared to its running in isolation (so higher is better).

$$WeightedSpeedup = \sum_{i=1}^{N} \frac{IPC_{shared}[i]}{IPC_{alone}[i]} \tag{5.1}$$

IPC_{alone} is the IPC of an application when it is running in isolation, IPC_{shared} is the IPC of the same application when it is running in conjunction with other applications, and N refers to the number of concurrent threads.

5.3.4 Comparison Approaches

To fully evaluate the partitioning scheme, it needs to be compared against four different approaches. *Unmanaged* is the baseline case. This corresponds to an LLC with no partitioning at all. Therefore, all cores compete for cache resources and can evict each others' data at any time. The next approach is *Fair Share* which corresponds to a statically-partitioned cache, where all cores have an equal number of ways, regardless of their memory behaviour.

CPE is a state-of-the-art static cache partitioning architecture for energy efficiency [114]. Using profile data, a static partition of the cache is computed. Applications can only access their designated regions of the cache, and these do not change during runtime. This design is the most flexible in terms of partitioning, because both sets and ways are configurable, leading to significant energy savings. In this comparison, the original architecture is extended to work with dynamic reconfiguration. This is achieved by profiling the applications and then using this data to drive the dynamic partitioning at runtime. Although unrealistic, this scheme serves as a useful comparison against an existing energy-focused technique.

UCP is a state-of-the-art dynamic cache partitioning scheme for high performance [109]. A Look-ahead algorithm is implemented in UCP to allocate ways to cores. Finally, *Cooperative Partitioning* is the proposed scheme which aims for high performance and large energy savings.

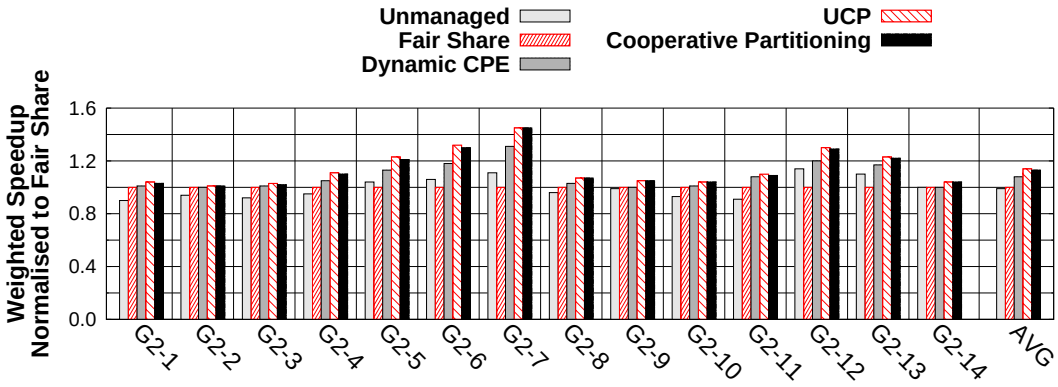


Figure 5.5: Weighted speedup of two-application workloads.

5.4 Evaluation

This section presents the evaluation of Cooperative Partitioning in terms of performance and energy consumption. Results are shown for both two-core and four-core systems. Unless otherwise stated, all results are normalised to the Fair Share scheme and the average used is the geometric mean.

5.4.1 Evaluation of a Two-Core System

5.4.1.1 Performance

Figure 5.5 shows the weighted speedup of each group of two-application workloads. It is clear that UCP and Cooperative Partitioning consistently have the highest performance across all combination of benchmarks. Further, almost all workloads benefit from some form of partitioning in the LLC. The exceptions are *Group2–5* to *Group2–7*, *Group2–12* and *Group2–13* where the Unmanaged cache performs better than the Fair Share scheme. Applications such as *astar*, *bzip2*, *gcc*, *perlbench* and *povray* benefit significantly from a large amount of cache space, which can be achieved in Unmanaged. In Fair Share, there are fixed boundaries which penalises these programs. Hence Unmanaged achieves a speedup of 14% in *Group2–12* because *gcc* is unconstrained. This motivates the need for a flexible cache partitioning approach, such as UCP or Cooperative Partitioning.

The modified comparison Dynamic CPE algorithm does not perform as well as would be expected, given that it has profile information to guide its partitioning decisions. This is because it has high flushing costs whenever altering the LLC configuration. When workload partitioning changes are infrequent, CPE performs close to

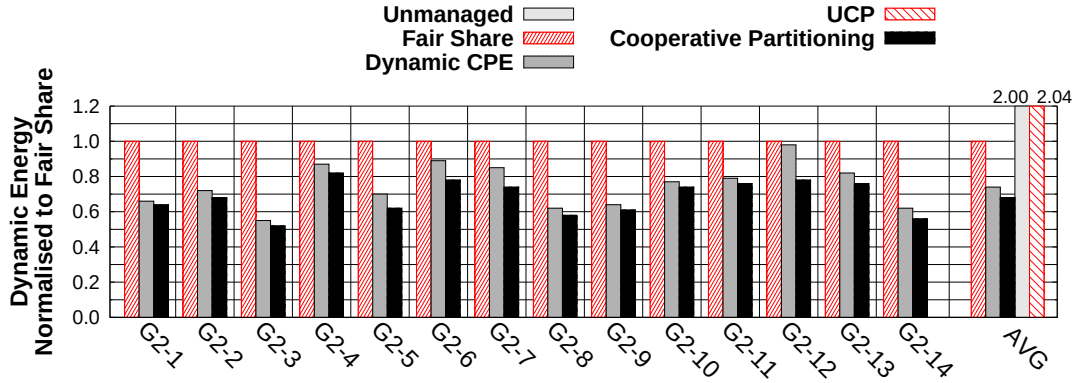


Figure 5.6: Dynamic energy consumption of the two-application workloads.

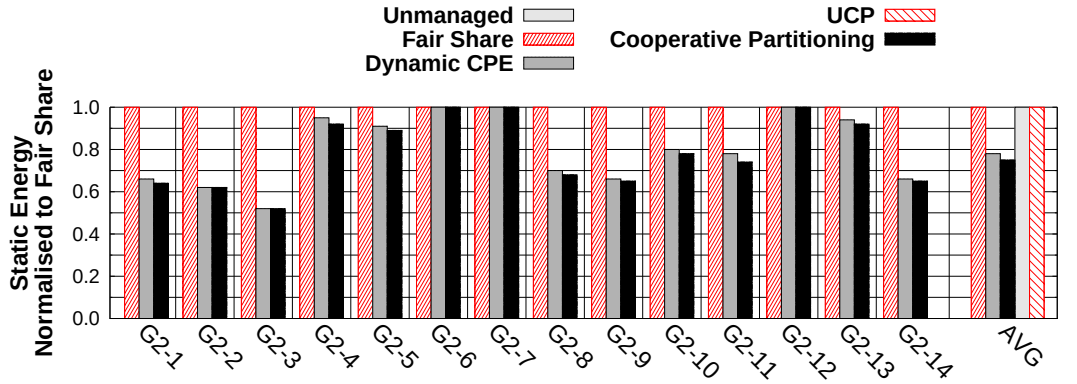


Figure 5.7: Static energy consumption of the two-application workloads.

UCP and the cooperative partitioning approach. This is most evident in workloads *Group2–1* to *Group2–3*. On the other hand, when there are frequent changes to the partitions, Dynamic CPE performs worse than UCP and Cooperative Partitioning. For example, in *Group2–7*, CPE achieves a speedup of 1.31, compared to 1.45 for UCP and Cooperative partitioning scheme, meaning it is 11% faster.

The performance of the cooperative partitioning approach is close to UCP. On an average, cooperative partitioning achieves a speedup of 1.13 and UCP achieves 1.14. The reason for is that it uses cooperative takeover of ways and must keep data way-aligned, whereas UCP does not have this restriction. As cooperative partitioning results show, in practice this is not a significant issue and it can still get large performance benefits despite this method of partitioning.

5.4.1.2 Dynamic Energy

Figure 5.6 shows the dynamic energy consumption of each different partitioning scheme. As explained earlier, Unmanaged and UCP do not provide dynamic energy savings as

they do not support aligned data (instead each access consults all cache ways). The cooperative partitioning approach achieves energy savings of up to 50% compared with the Fair Share scheme. This is because the cooperative partitioning scheme accesses only 2.9 ways, on an average, compared to 8 for the baseline and 4 for Fair Share. The largest savings are achieved by *Group2–3*, the reason being that on an average only two ways per access are active.

In *Group2–4*, *Group2–6*, *Group2–7*, *Group2–12* and *Group2–13*, frequent partitioning occurs due to the changing requirements of *astar*, *bzip2*, *gcc* and *povray*. For these workloads, CPE incurs significant overheads from flushing data while partitioning. However, the cooperative partitioning scheme can cope with these changes and still achieve significant energy savings (between 18% and 26%). On an average, Cooperative Partitioning has a dynamic energy consumption of just 68% of the Fair Share scheme, compared to 74% for CPE.

5.4.1.3 Static Energy

Static energy consumption is shown in Figure 5.7 and cooperative partitioning provides significant savings. The Unmanaged, UCP and Fair Share schemes do not reduce static energy because they do not enforce way-aligned data. In Cooperative Partitioning, when workloads under-utilise the cache memory, then the remaining cache ways can be turned off. In CPE, sets and ways can be shut down for static energy savings. As can be seen from Figure 5.7, in *Group2–2*, static energy savings of 48% are achieved. In this workload, only two ways are required by each application, on an average, therefore almost half the cache can be power-gated. However, for *Group2–6*, *Group2–7* and *Group2–12* the cache is used in its entirety, hence no ways can be turned off at all.

The unrealistic Dynamic CPE scheme also saves considerable amounts of energy, although never more than Cooperative Partitioning. On average the cooperative partitioning approach consumes 75% of the static energy of the Unmanaged, UCP and Fair Share caches, whilst Dynamic CPE consumes 78%. Overall, Cooperative Partitioning consumes less energy than other schemes, with performance just 1% away from the best across all comparison approaches.

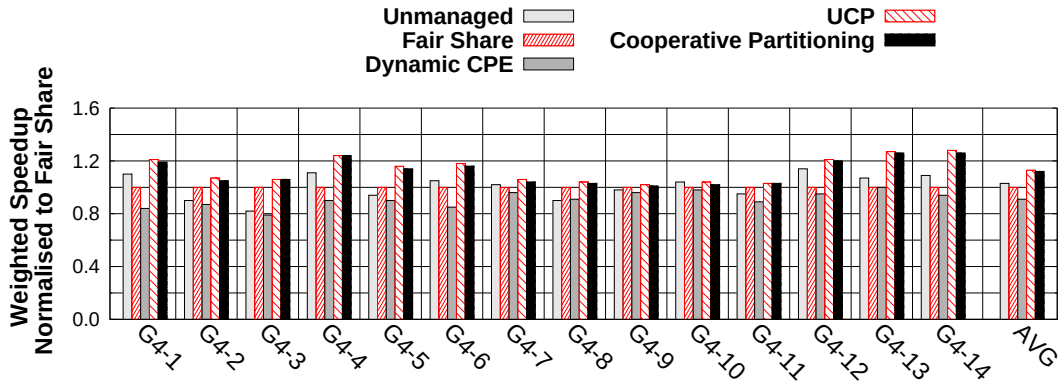


Figure 5.8: Weighted speedup of the four-application workloads.

5.4.2 Evaluation of a Four-Core System

5.4.2.1 Performance

Figure 5.8 shows the weighted speedup of the four application workloads, where it is clear to see that Dynamic CPE performs very poorly. This is due to frequent partitioning changes, leading to significant amounts of flushing in this approach. Dynamic CPE is not scalable across a large number of cores because the number of flushes increases with the number of applications.

Workloads like *Group4–3* have small cache requirements, meaning that there is not a significant amount of performance improvement available beyond the Fair Share scheme. Further, these workloads benefit significantly from partitioning due to thrashing between two applications (*sjeng* and *soplex*) in Unmanaged. On the other hand, workloads like *Group4–13* contain at least one application that requires a large fraction of the cache (i.e., more than a quarter given by Fair Share). In this case the program is *gcc* which obtains 7 ways on average. Fair Share unnecessarily constrains these applications, meaning that other schemes can achieve significant speedups.

As in the two-application workloads, Cooperative Partitioning performs similarly to UCP and is never slower than Fair Share. On an average, UCP achieves a 1.13 speedup whereas cooperative partitioning approach achieves 1.12.

5.4.2.2 Dynamic Energy

Figure 5.9 shows the dynamic energy consumption of these workloads. *Group4–3* obtains the lowest energy consumption that is 46% of the Fair Share scheme. In this workload, two applications get only two ways within the cache and these account for the majority of LLC accesses. Workloads *Group4–4*, *Group4–12* and *Group4–13* have

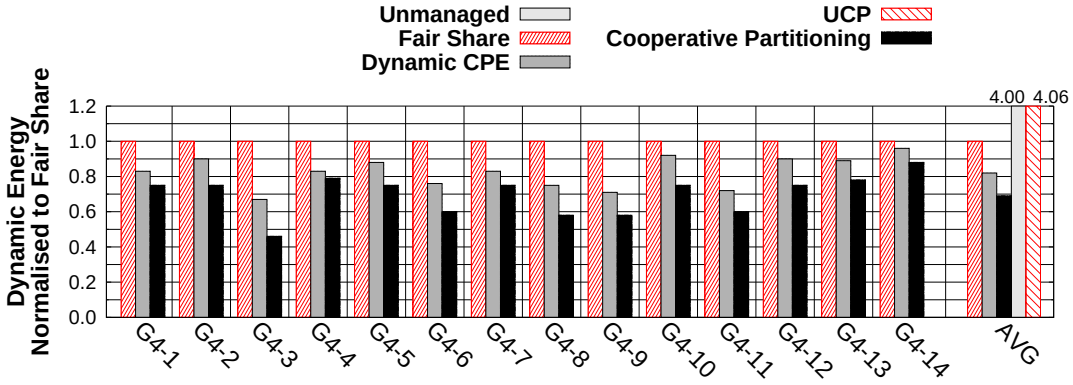


Figure 5.9: Dynamic energy consumption of the four-application workloads.

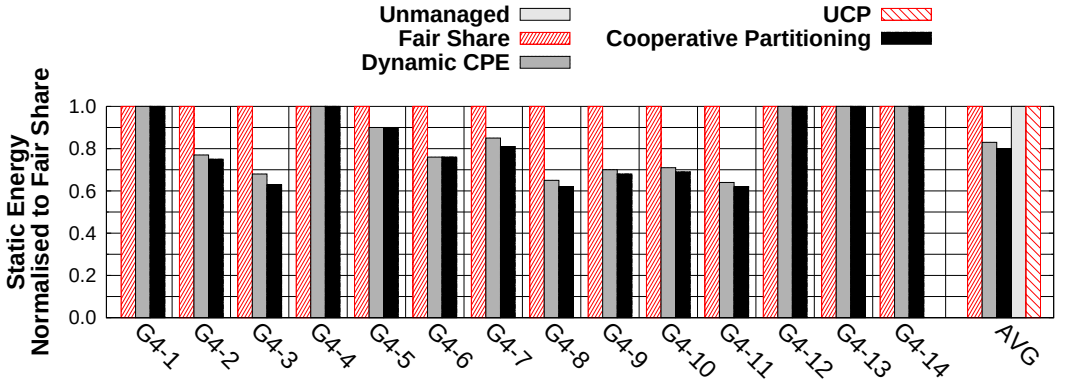


Figure 5.10: Static energy consumption of the four-application workloads.

at least one application that benefits from a large LLC. These are *astar* and *gcc*. Therefore it might be reasonable to expect the dynamic energy consumption of these groups to be larger in Cooperative Partitioning than in Fair Share, because these applications are assigned larger cache partitions which consume more energy on each access. However, the energy increases prove to be negligible compared to the consumption from the high MPKI applications that co-execute alongside. The memory intensive applications get assigned to a narrow partition, so that it consumes less energy than in Fair Share. Since these dominate the cache accesses, cooperative partitioning scheme ends up with significant dynamic energy savings, even in these cases.

In total, cooperative partitioning approach consumes just 69% of the dynamic energy of the Fair Share scheme. In comparison, Dynamic CPE consumes 82%. This is because cooperative partitioning accesses 3.2 ways on an average, compared to 4 for Fair Share.

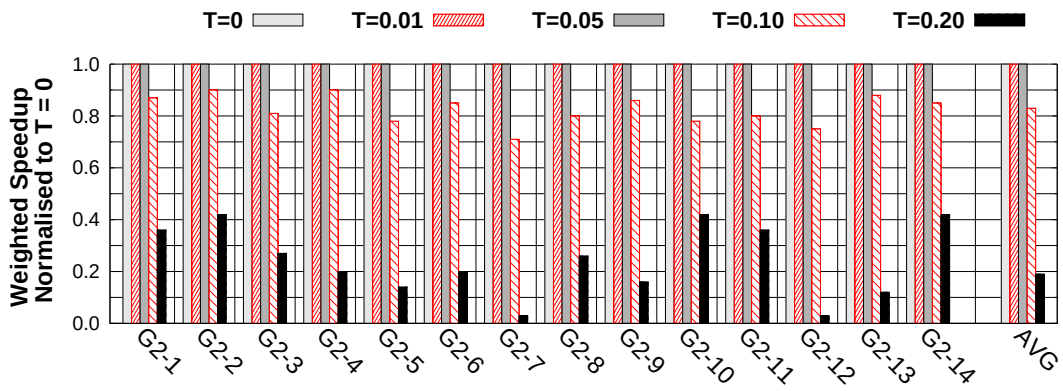


Figure 5.11: Impact of altering the takeover threshold value on performance.

5.4.2.3 Static Energy

Finally, static energy consumption is shown in Figure 5.10. Here, five workloads completely utilise the cache space, meaning that no ways are turned off. However, in the other groups, large savings are achieved by Cooperative Partitioning, such as *Group4–3*, *Group4–8* and *Group4–11* where there are 38% savings. This is because these workloads use fewer ways on an average, compared to other schemes (e.g., two applications have 2 ways and two have 3 ways in *Group4–11*). This leads to an average static energy consumption of 80% of the Fair Share approach.

5.5 Analysis of Results

Having evaluated cooperative partitioning in terms of both performance and energy consumption, this section now analyses the reasons for the benefits seen. First, it considers the sensitivity of the algorithm to the turn-off threshold, to show how small values maintain high performance but enable large energy savings. It then shows the amount of time taken to transfer ways between cores, which has to be as short as possible. To show how cooperation between cores takes place when migrating ways, this section shows the types of access that set the takeover bit vector, then analyses the LLC to memory bandwidth used when transferring. Also, this section conducts the analysis on the two-application workloads only, due to space limitations. In addition, the four-application workloads behave similarly and thus the same conclusions can be applied to them.

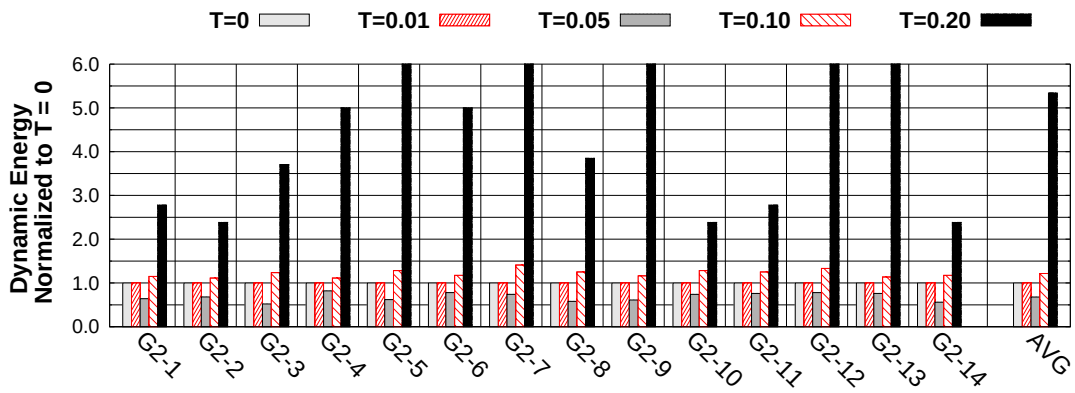


Figure 5.12: Impact of altering the takeover threshold value on dynamic energy.

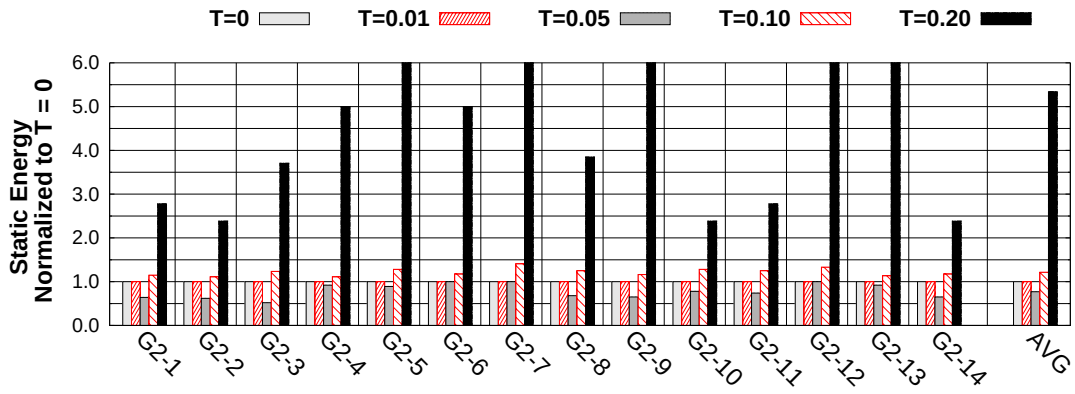


Figure 5.13: Impact of altering the takeover threshold value on static energy.

5.5.1 Impact of Takeover Threshold

Figure 5.11 shows the performance impact of altering the takeover threshold, described in Section 6.2.3, for a range of thresholds, from 0 to 0.2. A threshold value of 0 corresponds to an allocation of ways in the same manner as UCP. Increasing it makes it more difficult for an application to obtain more ways; they are only given out if the application significantly benefits from them. At the other extreme, a threshold value of 1 would mean that no ways were ever allocated to any core.

When a threshold value is 0.05 or less, there is no change in performance compared with a threshold of 0. For a 0.1 threshold, 17% performance loss is incurred. When this is increased to 0.2, all workloads experience large performance losses. The reason is that with such a high value, performance benefits from increasing ways are less than the threshold allows. Therefore, the algorithm falsely prohibits the acquisition of extra ways, leading to poor performance.

On the other hand, large energy savings can be achieved as the threshold value increases. These are shown in Figures 5.12-5.13. With a threshold of 0.05, almost

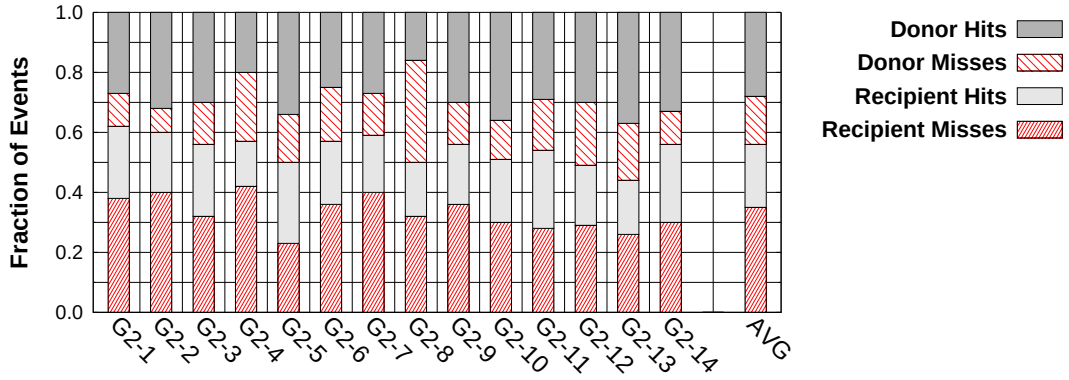


Figure 5.14: Events that set takeover bits when transferring ways between cores.

all workloads achieve dynamic energy savings (all apart from *Group2–6*, *Group2–7* and *Group2–12*) and all achieve static energy savings. This justifies the use of a 0.05 threshold value for all other experiments, as this provides a good trade-off between high performance and significant energy savings.

5.5.2 Cooperative Takeover Events

Cooperative Partitioning relies on cooperative takeover to implement its partitioning decisions. Figure 5.14 shows the breakdown of events that set takeover bits when transferring ways between cores. It shows hits and misses by the donor and recipient cores for each of the fourteen workloads.

In almost all groups, donor hits and recipient misses account for well over half the takeover bits being set. In the majority of cases, these events correspond to approximately two-thirds of the bits being set. The only exception is *Group2–8* where this only happens 48% of the time.

There is an intuitive reason for this finding. The donor core has enough space in the LLC and, in fact, is giving away one of its ways because it does not need so much room. Therefore most of its accesses will hit in the cache. On the other hand, the recipient core needs more space because its data cannot fit comfortably in the LLC. Therefore, it will miss frequently in the cache until its allocation of ways increases. Hence, these two events are expected to be the most common, and, as Figure 5.14 shows, in practice they do lead to the majority of takeover bits being set.

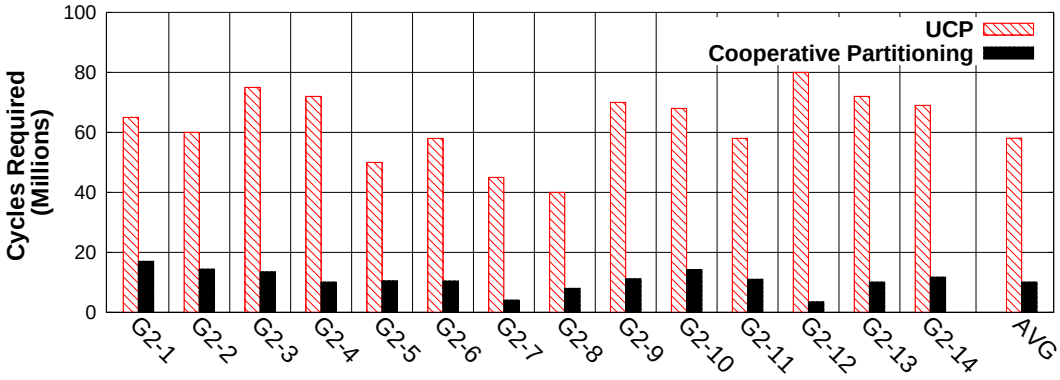


Figure 5.15: Cycles taken to transfer a way in a two-core system.

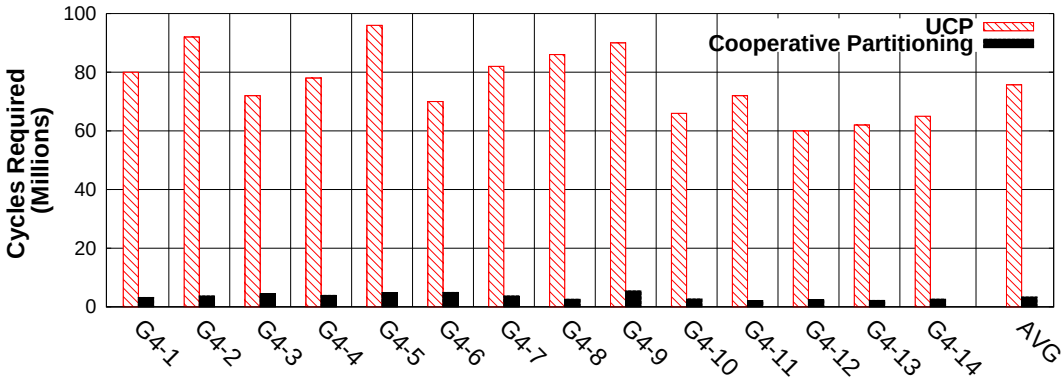


Figure 5.16: Cycles taken to transfer a way in a four-core system.

5.5.3 Transition Time

Setting of takeover bits on donor and recipient accesses means that ways can quickly be transferred between cores. To quantify the amount of time this actually takes, consider Figure 5.15 and Figure 5.16. This shows the average number of cycles for Cooperative Partitioning to transfer each complete way between cores for each workload. For comparison, UCP is also shown. Since UCP does not enforce way-aligned data, this value corresponds to the average number of cycles taken to transfer one block from each set.

It is clear that Cooperative Partitioning is significantly faster to transfer ways than UCP. On an average, in a two-core system, it takes 10M cycles whereas UCP takes 58M and in a four-core system, cooperative partitioning takes 3.3M cycles whereas UCP takes 75M cycles. Since these account for just 33% of all accesses to each way during partitioning (as shown in Figure 5.14), it follows that Cooperative Partitioning is faster. Further, there are some blocks that are infrequently accessed and these take a large number of cycles to cause a recipient miss. This also accounts for the large transition time in UCP.

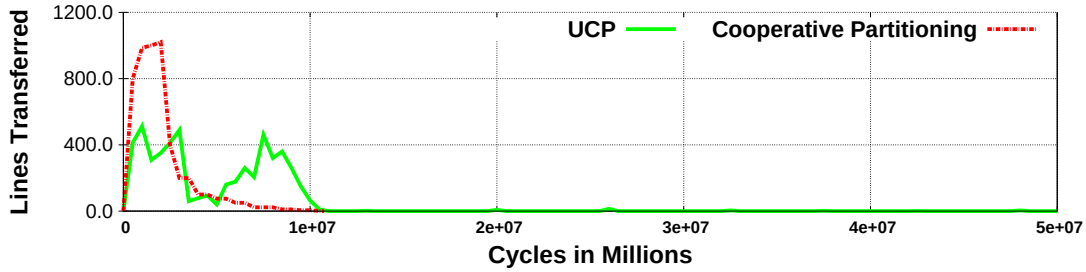


Figure 5.17: LLC to memory bandwidth usage for flushing data after a partitioning decision in a two-core system.

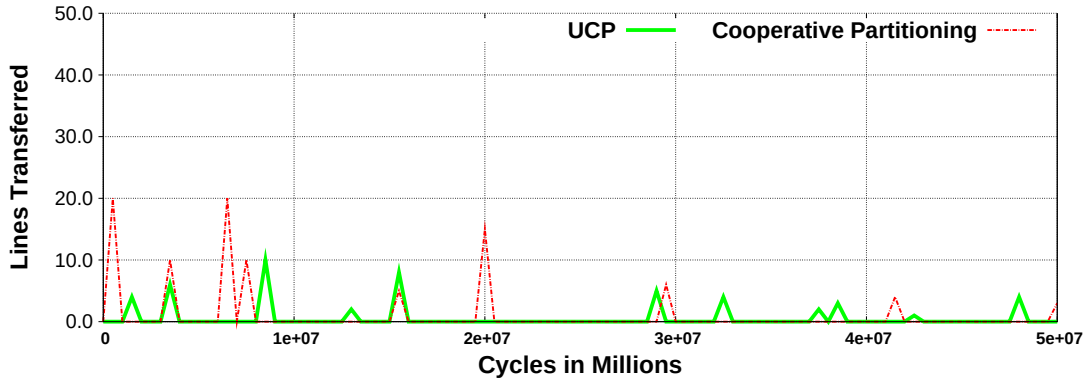


Figure 5.18: Memory to LLC bandwidth usage after flushing the data in a two-core system.

5.5.3.1 Reconfiguration Frequency

Also it should be noted that, on an average, the reconfiguration frequency is, once in every 50 Million cycles for a two-core system and once in every 25 Million cycles for a four core system. Thus, on an average, 20% of time is spent in transition for a two-core system and 12% of time is spent in transition for a four core system.

5.5.4 Memory Bandwidth Usage

The final analysis concerns the amount of memory bandwidth used to flush dirty cache blocks back to main memory during a transition. Figure 5.17 and Figure 5.19 show how the average number of flushed blocks vary over time once a partitioning decision has been made. Due to the speed that Cooperative Partitioning transfers ways between cores (Section 5.5.3), it incurs a higher cost initially, with a large number of lines being flushed. In a two-core system, this quickly drops off 6 million cycles after a partitioning decision is made and stays close to 0 until 10 million when the transfer of

the way is complete. In a four-core system, this quickly drops to zero at 3.3 million cycles after a partitioning decision is made.

In a two-core system, for UCP, there is also a peak in the number of lines flushed during the first period (up to 4 million cycles in a two-core system and 1 million cycles in a four-core system), although its magnitude is considerably smaller as compared to Cooperative Partitioning. However, after this point there is a steady use of memory bandwidth, rising to another peak at 7.5 million cycles in a two core system, then down to nearly 0 just after 10 million cycles. As previously stated, UCP does not complete its transfer until 58 million cycles and 78 million cycles in a two-core and four-core system respectively have passed. Therefore UCP has a more constant memory bandwidth usage, whereas Cooperative Partitioning causes a larger activity burst in a short period of time.

In fact, what is not shown is that UCP has to flush more lines from LLC to memory during a transition than Cooperative Partitioning. This is because UCP only flushes on a miss by the recipient. Before this happens there can be multiple writes by the donor core, making clean blocks dirty. Although this can also happen in Cooperative Partitioning, it is much less likely, since all accesses by the donor will cause the takeover bit vector to be set and the block transferred to the recipient. On an average, Cooperative Partitioning flushes 5102 lines, whereas UCP flushes 6536.

Having shown the traffic from LLC to memory, it will be interesting to show the traffic from memory to LLC. This will give more information, since cooperative partitioning selects random data from the ways as it might end up in selecting the MRU line. Figure 5.18 and Figure 5.20 show the total number of lines transferred from memory to LLC which implements UCP, cooperative partitioning scheme and a baseline unmanaged cache. Since UCP selects LRU line, the number of lines transferred back from memory to LLC is not that much, whereas, in cooperative partitioning scheme, it picks the random lines from the ways, it does end up in transferring some extra blocks from main memory to LLC, but this is far less (less than 1%) to the baseline scheme.

5.6 Summary

This chapter has proposed Cooperative Partitioning, a novel partitioning scheme for last-level caches in CMPs. This approach maintains high performance while saving significant dynamic and static energy. It achieves this by enforcing way-aligned data within the cache, and by cooperation between cores when migrating ways between

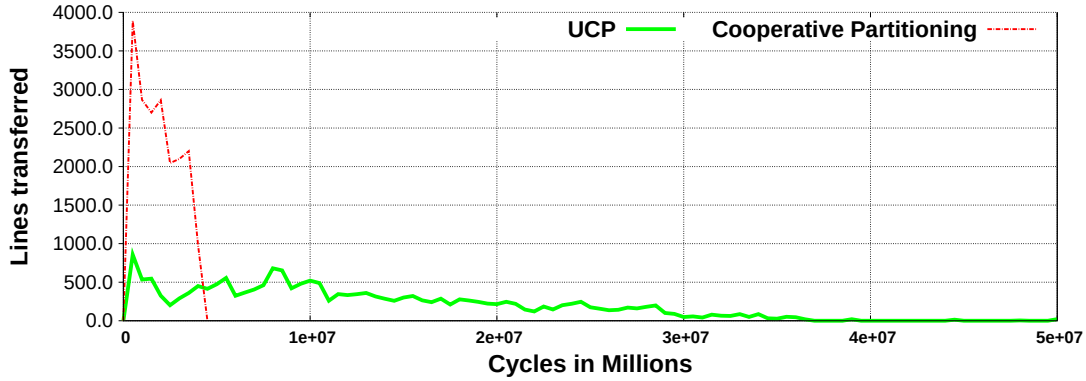


Figure 5.19: LLC to memory bandwidth usage for flushing data after a partitioning decision in a four-core system.

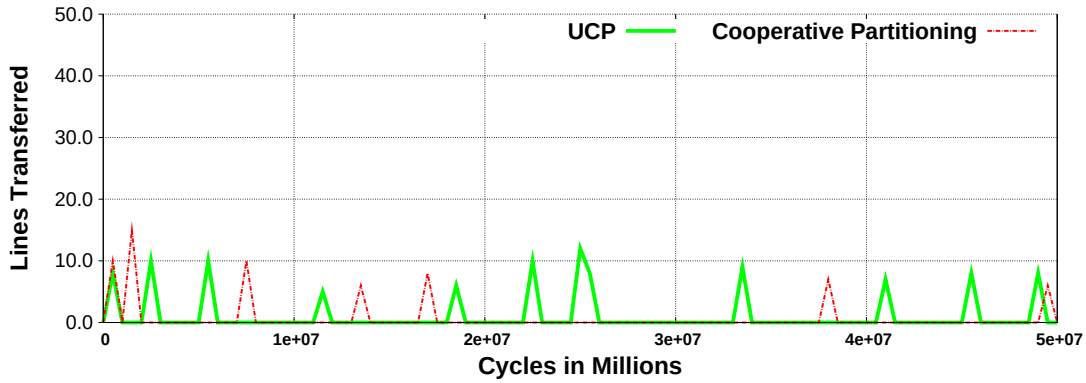


Figure 5.20: Memory to LLC bandwidth usage after flushing the data in a four-core system.

themselves. The state-of-the-art cache partitioning along with the fixed partitioning is implemented in the simulation infrastructure described in Section 5.3. Evaluation on a two-core system shows savings of 32% dynamic and 25% static energy compared to a fixed partitioning scheme. In a four-core environment, dynamic and static energy savings of 31% and 20% are achieved, with negligible loss of performance. Further, cooperative partitioning scheme migrates ways between cores five times more quickly than a state-of-the-art partitioning approach and requires less data to be flushed back to memory.

The energy savings realised by Cooperative Partitioning create additional headroom in the processor's thermal design power. Thus the negligible reduction in performance can be mitigated through higher clock rates for the same number of cores.

Chapter 6

Region Aware Cache Partitioning

In recent years, high performance computing systems have obtained more processing cores and share a last level cache (LLC). However, as their number grows, the core-to-way ratio in the LLC increases, presenting problems to existing cache partitioning techniques which require more ways than cores. Further, effective energy management of the LLC becomes increasingly important due to its size.

Most previous work on cache energy saving for LLC, including the solution proposed in Chapter 5, holds good when the core to cache way ratio is 1:4. This ratio means that for a two core system, the LLC will have 8 ways, similarly for a four core system, the LLC will have 16 ways. However this ratio does not hold good for many core systems, which have a core to way ratio of 1:2 or 1:1. This chapter proposes a Region Aware Cache Partitioning (RECAP), an LLC energy-saving scheme for high-performance, many-core processors, that has the core to way ratio of 1:2 or 1:1.

RECAP partitions the data within the cache into shared and private regions. Applications only access the ways containing the type of data that they require, realizing dynamic energy savings. Any ways that are not within the shared or private regions can be turned off to save static energy. The proposed partitioning scheme is evaluated using an 8-core CMP running multi-programmed workloads and shows that it achieves 68% dynamic and 33% static energy savings in the shared LLC with no performance loss. Across the multi-threaded applications, RECAP achieves 77% dynamic and 60% static energy savings, again with no slowdown.

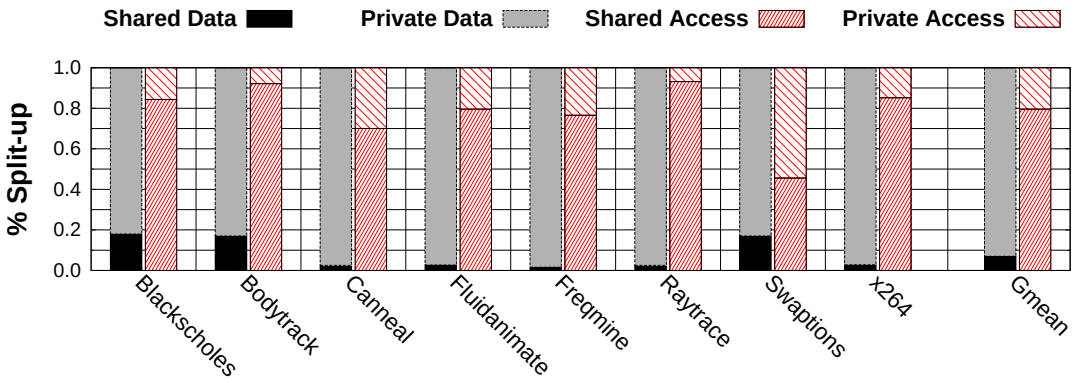


Figure 6.1: Data types within and accesses to a shared last level cache.

6.1 Introduction

The last level cache (LLC) within a chip multiprocessor (CMP) is the largest cache within the processor and is often shared among all cores on the chip. It accounts for a significant fraction of the total chip power budget, therefore decreasing its energy consumption realizes significant system benefits. However, it is a critical structure for performance, standing between the rest of the processor and long-latency main memory. Therefore, any energy-saving scheme for the LLC should cause minimal or no performance degradation.

There is a rich body of work in the literature proposing cache energy reduction techniques, although, till date, the majority of existing research has focused on single-core systems. These schemes reduce static energy consumption by turning off parts of the cache, or achieve dynamic energy savings by predicting the ways that should be accessed for each load or store. However, the filtering effects of the higher-level caches in a CMP make the access patterns hard to predict, meaning that these schemes are not suitable for a multi-core scenario.

In the last ten years, cache partitioning for performance has received significant interest [124, 125, 136, 137], the idea being to give each core a share of the cache resources according to their needs. These schemes have the potential to realize significant performance increases, yet for the most part they do not consider energy saving within the LLC at all. Further, to work effectively they often require the number of ways to be more than the number of sharing cores.

This work takes a different approach. Instead of partitioning based on the requirements of each core, RECAP partitions based on the *type of data* that is being accessed — shared or private. Consider Figure 6.1 which shows the number of accesses to

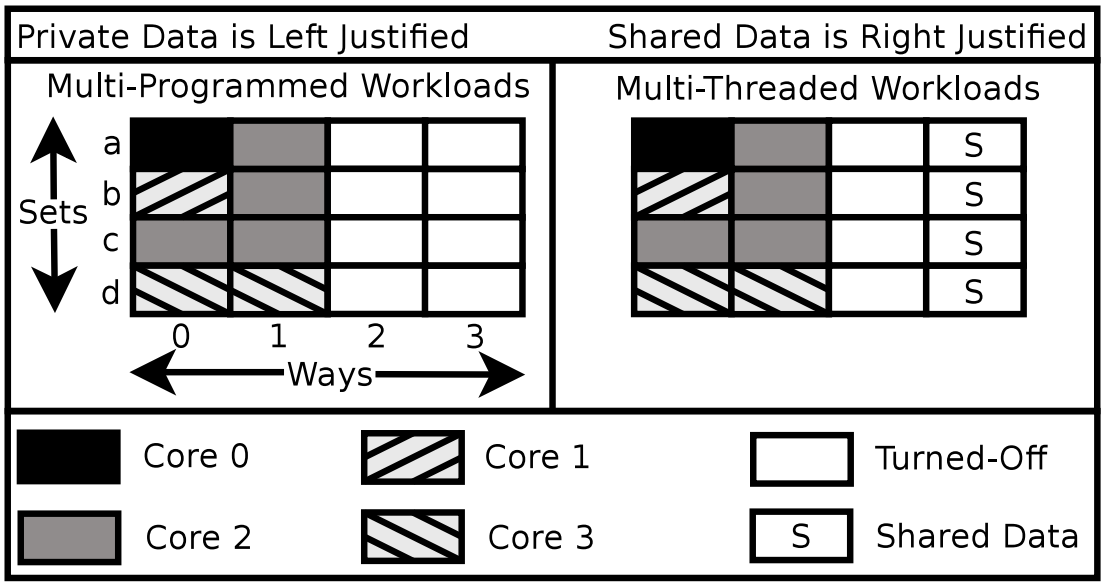


Figure 6.2: Example data layouts for differing workloads in RECAP.

shared and private data within the LLC for the Parsec benchmark suite and the amount of shared and private data resident in the cache for the system described in Section 5.3. The overwhelming majority (80%) of accesses request shared data, yet the vast majority of data in the cache is actually private (93%). Other researchers have seen similar statistics [27].

This motivates Region-Aware Cache Partitioning architecture (RECAP), which is shown in Figure 6.2. RECAP identifies and separates private and shared data, justifying the former to the left-hand side of the cache and the latter to the right. When an access requires private data, only the ways on the left of the cache need to be activated, and vice-versa for shared requests. This way alignment of data allows us to realize significant dynamic energy savings. Further, RECAP monitors the cache usage of each core in the system and dynamically restricts the size of the private region for each one. This means that ways in the centre of the cache become unused and can be turned off for static energy savings. In the literature, RECAP was the first to use separate partitioned cache regions for private and shared data.

Using RECAP's technique in the simulation infrastructure described in Section 6.3, for an 8-core system with an 8-way cache achieves dynamic energy savings of 68% and static energy savings of 33% with no performance loss across a range of multi-programmed workloads. For multi-threaded applications, RECAP achieves 77% dynamic and 60% static energy savings at the same level of performance. Further, RECAP introduces a simple scheme whereby all cores help in flushing dirty data back to

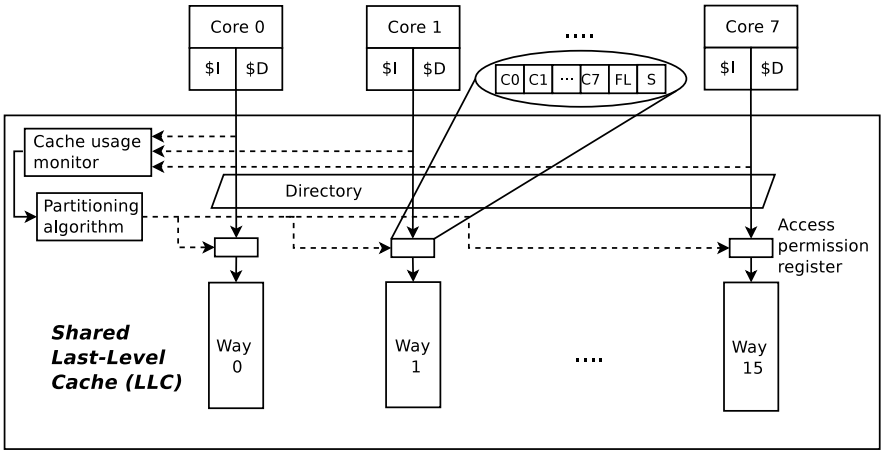


Figure 6.3: An overview of the RECAP architecture. Cache use by each core is monitored and periodic partitioning decisions are made. The ability of each core to read or write into each way is granted through access permission registers (APRs).

main memory when a core no longer requires a cache way, and shows that this is 85% faster and consumes 98% less energy than a basic technique that relies on a single core alone.

The rest of this section is structured as follows. Section 6.2 presents RECAP, then Section 6.3 describes the experimental methodology. Section 6.4 evaluates RECAP approach on an 8-core systems and analyses the results. Section 6.5 concludes.

6.2 The RECAP Architecture

There are two parts to RECAP. The first is the monitoring of cache usage for each individual core. This allows us to determine the optimal number of cache ways to assign to each core to maintain high performance but enable significant energy reduction. The second part actually implements the way assignments and realizes these energy benefits. This chapter assumes a serially-accessed last level cache, as is the norm. Therefore dynamic energy savings come from the tag side only.

Figure 6.3 gives an overview of RECAP. All accesses from the higher-level caches pass through the directory on their way to the LLC. Over an interval containing a fixed number of instructions, these accesses are monitored by the cache usage logic and are then used to make partitioning decisions. These are written into the access permission registers (APRs) that sit before the cache ways, to control which cores have access to each way. Data that is private to a particular thread is kept left-justified in the cache (i.e., in the low-numbered ways) whereas shared data is kept right-justified (i.e., in the

high-numbered ways).

The proposed cache architecture is orthogonal to existing cache partitioning schemes [109, 129, 136] because these techniques consider cache partitioning only when the number of ways is greater than or equal to the number of cores. RECAP cache partitioning is performed whatever the core-to-way ratio is. Further, RECAP approach can be applied to any LLC configuration, from banked NUCAs to multiple, independent LLCs.

This chapter first describes the cache monitoring scheme, then explains access permission registers in more detail. Then it describes how to reconfigure the ways that each core can access, as well as how to keep private and shared data separate. Finally, the overheads associated with RECAP cache architecture are explained.

6.2.1 Usage Monitoring

RECAP cache architecture builds on prior work to determine the optimal number of ways for each core. RECAP uses utility monitors [109] to track the accesses by each core to characterize each thread's use of the cache. These monitors have been used by other partitioning techniques [129, 136]. However, the operating system could also provide this information [112].

Algorithm 3 is used to determine the partitions. RECAP is not concerned with allocating ways to each core individually, so it only needs to find the number of ways for each core to achieve its highest utility (max_mu). RECAP assumes all threads have equal priority and therefore gives each core a number of ways based on the performance benefits it can realize from them. If threads were to have differing priorities, RECAP can incorporate this information into Algorithm 3 by increasing the number of ways that high-priority threads receive and decreasing the number of ways allocated to low-priority threads.

6.2.2 Cache Partitioning Control

To control each core's accesses to the LLC, and to enforce way-aligned data, RECAP introduces an access permission register (APR) to each way. The layout of these registers is shown in Figure 6.3. Each APR has one bit per core to indicate whether a core has access permission to the associated way. There is an additional shared bit to indicate a way shared by all cores and a further flush bit which is used when contracting a core's cache allocation. The process of contraction is explained in Section 6.2.3.

Algorithm 3: Determine the cache requirement for each core.

```
T_Ways = N; /* Total number of cache ways in LLC*/
blocks_req[c] = 0; /* For each competing core, c */
foreach core c do
    max_mu[c] = get_max_mu(c, T_Ways);
    blocks_req[c] = min blocks to get max_mu[c] for core c;
return blocks_req;

get_max_mu(c, T_Ways):
max_mu = 0;
for j = 1; j <= T_Ways; j++ do
    U = change in misses for core c when moving from 0 to j ways;
    mu = U / j;
    if mu > max_mu then
        max_mu = mu;
return max_mu;
```

For a given cache way, when a core's bit is set in the corresponding APR, it has permission to both read and write in that way. When it is unset, then the core cannot access that way at all. However, during contraction, the flush bit is set and this allows all threads to read from that way, regardless of their core bit in the APR. This leads to three possible modes of operation, with read access to a way controlled by *APR[id]* —— *APR[fl]*, and write permission by *APR[id]*, where *id* is the core's id and *fl* is the flush bit. With the core's APR bit set, the core can both read and write in that way. With the core's bit unset, but the flush bit set, the core can only read from the way. When both bits are clear, then the core cannot access that way at all.

Since the number of cache ways may be equal to or less than the number of cores, more than one core will have its APR bit set for certain ways. Thus RECAP provides pseudo way-alignment, in contrast to other schemes [129], utilizing the cache more effectively.

The APR register enforces the cache partitioning by only allowing cores to access certain ways. At the same time, this enables dynamic energy savings because cores only need to access the ways where they have permission. If the APR for a way is totally clear (i.e., all core bits, shared and flush bits unset), then the way can be turned off, realizing static energy savings.

Algorithm 4: Steps taken to reconfigure the cache.

Step 1: Initially all cores share the left-most way for private data and right-most way for shared data.

Step 2: At the end of each interval, run Algorithm 3.

Step 3: When a core requires more ways than in the previous interval, it expands its ways in the private region towards the right.

Step 4: When a core requires fewer ways than in the previous interval, it contracts out of the right-most ways in the private region.

Step 5: Repeat *Step 2* to *Step 4*.

6.2.3 Cache Reconfiguration

Algorithm 4 gives a high-level overview of the steps taken after each phase interval, having calculated the way requirements for each core with Algorithm 3. Initially all cores share one cache way (way 0) so all core bits in the APR for this way are set, enabling full access to all threads. When a core requires more cache ways than it currently has, RECAP enters an expansion mode for that thread. On the other hand, when a core needs fewer ways than it currently has access to, RECAP enters a contraction mode.

6.2.3.1 Expansion Mode

Expanding the number of ways that a core can access is simply a matter of setting the appropriate bit in the APR for each way that the core requires. If any of the ways were previously turned off, then they are enabled once more.

6.2.3.2 Contraction Mode

When a core requires fewer ways than it currently has, the cache cannot simply reset the core's APR bit for the ways it no longer needs. This is because the core may have dirty data in those ways which it still needs to read. Therefore, RECAP only prevents write access to those ways and flush dirty data back to main memory as each set is

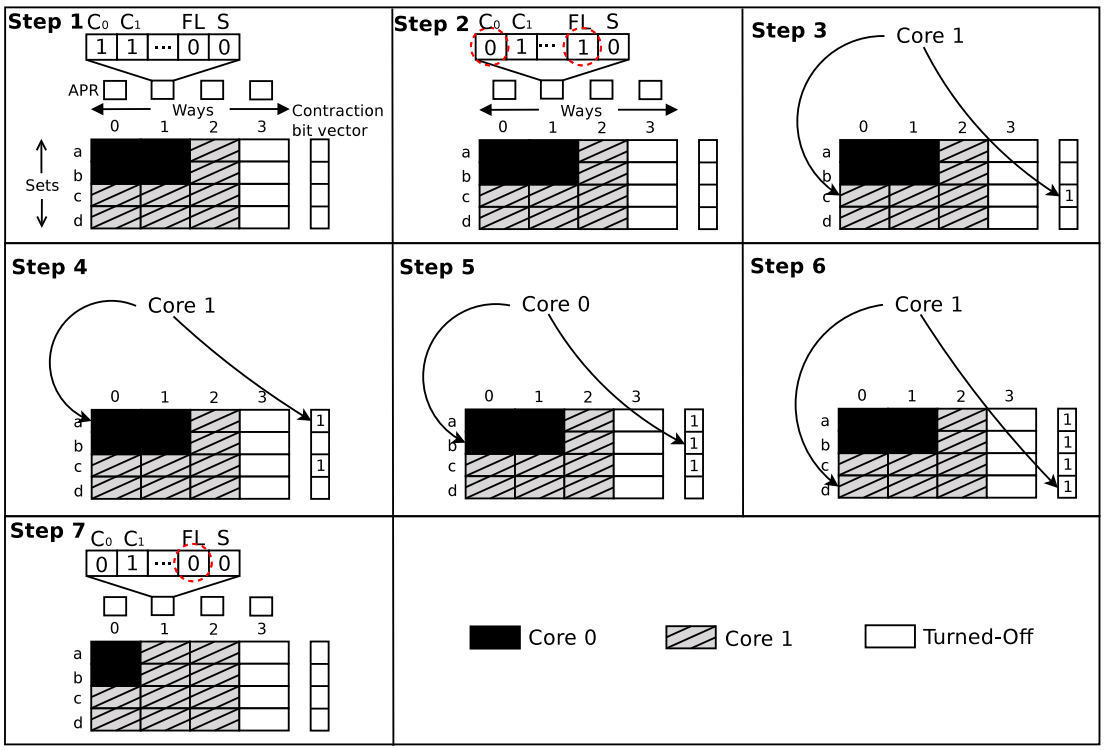


Figure 6.4: An example of contraction where core 0's data in way 1 is completely evicted.

accessed. To keep track of the sets that need to be flushed (if they contain dirty data), RECAP provides a bit vector for each way that is initially clear, with each bit being set according to the sets that are accessed. Once this contraction bit vector is totally set, no more dirty data from the contracting core exists in that way and all permissions can be removed.

One downside with this approach is that it can take many cycles for all sets to be accessed, which decreases the effectiveness of RECAP partitioning scheme and can lead to fewer energy savings (because it takes longer for unneeded ways to be turned off). To aid the flushing process, RECAP provides an extra bit in the APR, called the *flush bit*. When this is set, all cores access the corresponding way to flush data back to main memory from the set they are consulting, and set the relevant bit in the contraction bit vector. So instead of one core being responsible for all the flushing, all cores contribute to it, significantly speeding up the contraction process. Whenever a phase transition occurs during the takeover process, RECAP resets the bit vector and starts the whole process again.

6.2.4 Contraction Example

Figure 6.4 shows an example of contraction in practice. In this example there are two cores and four cache ways. Initially, both cores share ways 0 and 1, with core 1 also having full access to way 2, as shown in step 1.

However, after a partitioning decision has been made, core 0 requires just one way, whereas core 1 still needs all three of its ways. Core 0 therefore has to contract and stop using way 1, leaving it just for core 1.

To start this process, core 0's bit in the APR for way 1 is unset and the flush bit is set. This means that core 0 can no longer write into way 1 and both cores will participate in flushing core 0's dirty data back to main memory. Notice that the contraction bit vector for this way is clear, showing that no data has been flushed back.

Core 1 now accesses set c. Core 0 does not have any data in way 1 in this set, so no flushing needs to occur, and the corresponding bit in the vector is set to show this. Core 1 now accesses set a and this time some of core 0's data is in way 1, so it is flushed back to main memory and the relevant bit set. Core 0 accesses set b and also flushes its data back, setting the bit at the same time. Finally, core 1 accesses set d and sets the final bit in the vector.

The cache now recognizes that all bits in the contraction vector are set, so it resets the flush bit in the APR, meaning that contraction is complete. The contraction bit vector is also reset at this point, ready for the next contraction phase. Core 0 can now only access way 0, whereas core 1 can access ways 0 through 2.

6.2.5 Separating Private and Shared Data

One key contribution of RECAP, as highlighted in Section 6.1, is the ability to realize significant energy savings simply based on the type of data being accessed — private or shared. As such, RECAP must be able to identify these different types of data and deal with the situations where they change from one category to the other.

All accesses to the LLC first pass through the directory. RECAP uses the directory information to determine whether the required data is private or shared. Data that is in exclusive or modified states is considered private whereas data that is in a shared state is shared.

When a core accesses the LLC, it only searches in the ways that contain the type of data that it wants, i.e., when looking for private data it only searches the private ways that it has access to (with its core bit set in the APR). When searching for shared data

it only looks in the shared ways (those that have the shared bit set in the APR). When a directory does not have an entry for a particular data item, the core searches its ways in both the private and shared regions.

RECAP uses the default LRU replacement policy to evict data on a miss, requiring only minor changes to the baseline LRU scheme. Upon a core's access, the cache ways that have that core's bit set in their APR are enabled and the LRU policy is applied to the enabled ways to find the victim cache block.

There are two situations where RECAP has to alter the category of the data in the LLC. First, if the data is in the shared region but a core wants to write to it, RECAP simply invalidates it in the LLC as usual. If the core then evicts that data from its L1 cache (e.g., through a capacity miss), then the data will be written into the private region of the LLC. If, on the other hand, another core requests read access to that data, it will be written back into the LLC's shared side.

Second, if the data is in the private region, but another core wants to read it, then the data must be placed into the shared region. To do this, RECAP initially sends the block back the requesting core, then invalidates the existing line (flushing dirty data, if necessary) and pre-fetches from main memory, writing into the shared region. This incurs an overhead, but is more simple than implementing logic to move the line between two regions. In practice, this happens infrequently and thus overheads are small.

6.2.6 Overheads of RECAP

RECAP uses the same hardware as Qureshi and Patt [109] to monitor cache usage, as is required by other partitioning schemes [129, 136]. RECAP also requires extra bits for the APRs and contraction bit vectors, detailed in Table 6.1. There is one APR per cache way, and each contains a bit per core, a flush bit and a shared bit. There is one contraction bit vector per core and each has a bit per cache set. For an 8-core system with an 8MB, 8-way cache, this comes to a little over 128K bits.

The RECAP cache consumes more power than a standard last level cache, because it has the extra circuitry for monitoring and partitioning. All power overheads are included in simulated results in Section 5.4.

In terms of performance, there is a negligible overhead due to each core being restricted in the number of ways it has access to when replacing data. In other words, RECAP incurs capacity misses slightly more frequently and may evict data that will

Hardware Description	Details	Bits
APR	10×8	80
Contraction Bit Vectors	$16{,}384 \times 8$	131,072
Total		131,152

Table 6.1: Summary of the hardware overheads of RECAP scheme for an 8-core system with an 8MB, 8-way LLC.

Parameters	**Configuration**
Processor	8-Cores, 1 thread/core, in-order
Operating System	Linux-2.6.28.smp
L1 ICache	32kB, 64B lines, 4-way, 3 cycle lat.
L1 DCache	32kB, 64B lines, 4-way, 3 cycle lat.
Shared L2	8MB, 64B lines, 8-way, 15 cycle lat.
Coherence	MOESI CMP directory

Table 6.2: System configuration.

be reused shortly afterwards that would not have been evicted had RECAP had access to more ways. However, in practice, this has negligible impact.

6.2.7 Summary

This section has described region-aware cache partitioning (RECAP), a high performance, energy-efficient partitioning scheme for last level caches. RECAP identifies private and shared data using information from the directory, placing private data in the left-hand side of the cache and shared data in the right. Additional access permission registers (APRs) control each core's access to the cache ways. This ensures that data is way-aligned and that unused ways can be easily identified and turned off.

6.3 Experimental Methodology

This section describes the environment used to evaluate the proposed cache architecture.

Parsec Benchmarks			
Blackscholes	Bodytrack	Canneal	Fluidanimate
Freqmine	Raytrace	Swaptions	X264

Table 6.3: Parsec workloads, all using sim-large inputs.

6.3.1 Simulator

RECAP is implemented in gem5 [13]. Table 6.2 shows the configuration of the system. An x86-based in-order processor running an SMP-enabled Linux kernel version 2.6.28 is simulated and an 8-core system is modelled to fully evaluate the effects of sharing and partitioning the last level cache. All level 1 caches are private and all processors share a common level 2 cache. The cache configurations are similar to those used by others [91, 106]. Cacti [131] at 45nm is used to obtain energy information. Finally, as in prior work [109], a 5 million cycle phase interval for monitoring and partitioning decisions is assumed.

6.3.2 Workloads

Parsec benchmarks [12] are used for the multi-threaded applications and a random mix of benchmarks from SPEC CPU2006 [122] for the multi-programmed workloads. Tables 6.3 and 6.4 show the applications. To choose the application mixes for SPEC, RECAP groups benchmarks based on their misses per kilo instructions (MPKI) values, denoting those with a value greater than 5 as *thrashing* applications. Then randomly selected mixes of 8 benchmarks containing 1 through 7 thrashing applications is used. So, in Table 6.4, 3T-1 refers to the first group containing 3 thrashing applications. Programs are ordered in Table 6.4 so that the thrashing applications occur first.

The *sim-large* inputs are used for all Parsec applications, simulating the parallel region in full. For the SPEC workloads *reference* inputs are used. Each application is fast-forwarded by 20 billion instructions, warmed-up the caches and branch predictor for 500 million instructions and then simulated for at least 1 billion further instructions, per application, as is common practice [44, 136]. Statistics are reported for 1 billion instructions per benchmark, but all applications continued running until the last program in the mix had reached this instruction count, to keep contending for cache resources.

Group	Benchmarks
1T-1	Mcf, Astar, Calc., Povray, Sjeng, Xalan, DealII, H264.
1T-2	Art, Bzip2, Gobmk, Namd, Omn., Perl., Gromacs, Xalan
1T-3	Equake, Omn., Astar, Gromacs, Gobmk, H264., Sjeng, Bzip2
2T-1	Sphinx3, Art, Gcc, Bzip2, DealII, Gobmk, Xalan, Povray
2T-2	Soplex, Lbm, Astar, Gobmk, H264., Namd, Sjeng, Perl.
2T-3	Mcf, Milc, Omn., Gcc, Xalan, Sjeng, Calc., Gromacs
3T-1	Mcf, Lbm, Milc, Bzip2, Namd, Povray, Sjeng, H264.
3T-2	Equake, Sphinx3, Libq., Omn., Sjeng, Gromacs, Astar, Gobmk
3T-3	Lbm, Libq, Soplex, Astar, Calc., Bzip2, Sjeng, H264.
4T-1	Equake, Sphinx3, Soplex, Lbm, Calc., Gromacs, Gcc, Gobmk
4T-2	Mcf, Milc, Libq., Art, Astar, Bzip2, Perl., Xalan
4T-3	Art, Equake, Sphinx3, Libq., Gcc, Omn., Namd, Povray
5T-1	Mcf, Lbm, Milc, Soplex, Libq., Gcc, Sjeng, Xalan
5T-2	Equake, Mcf, Lbm, Sphinx3, Art, Astar, Povray, Calc.
5T-3	Milc, Equake, Sphinx3, Libq., Lbm, Omn., DealII, Namd
6T-1	Mcf, Equake, Lbm, Libq., Art, Milc, Gcc, Sjeng
6T-2	Equake, Sphinx3, Mcf, Art, Soplex, Lbm, Astar, Namd
6T-3	Art, Milc, Lbm, Libq., Soplex, Equake, Gromacs, H264.
7T-1	Mcf, Lbm, Milc, Soplex, Libq., Art, Sphinx3, Povray
7T-2	Equake, Sphinx3, Art, Libq., Lbm, Milc, Soplex, Bzip2
7T-3	Art, Equake, Mcf, Soplex, Sphinx3, Lbm, Libq., Gcc

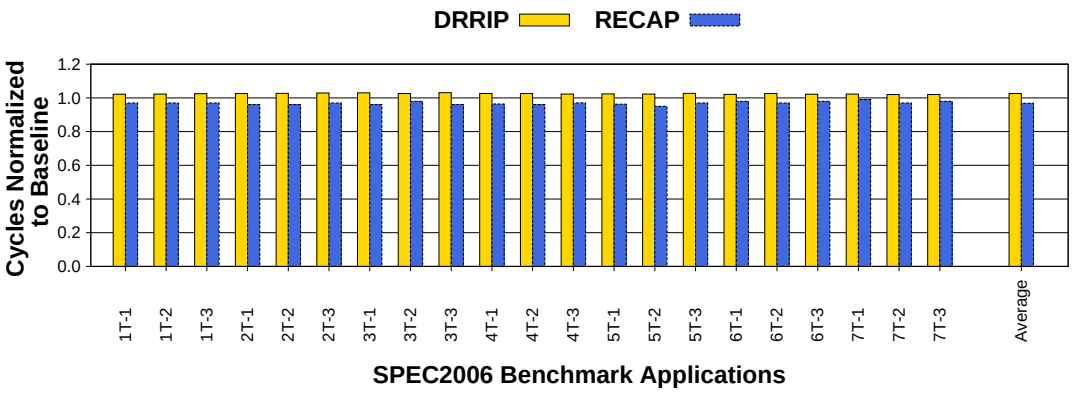
Table 6.4: SPEC2006 combinations, all using reference inputs.

6.3.3 Comparison Approaches

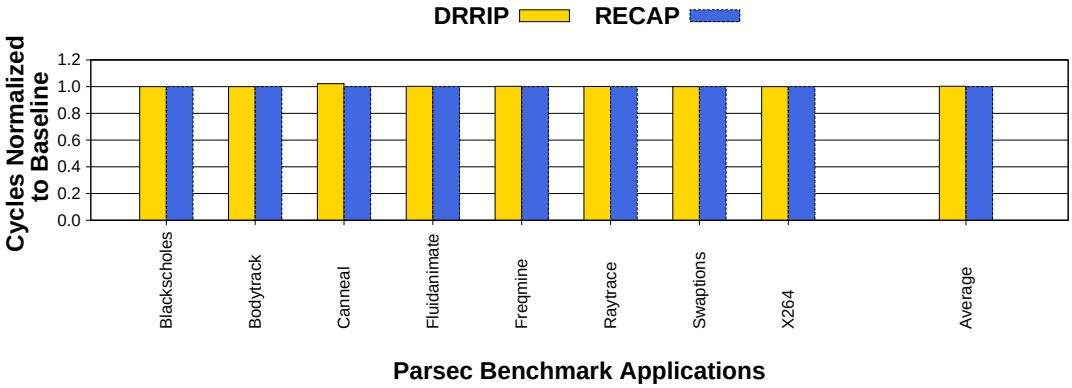
One significant problem with implementing comparison approaches is that existing schemes for cache partitioning only work when there are more cache ways than there are cores sharing that cache. RECAP is flexible and scalable, so is independent of the core to cache way ratio. Therefore, only one comparison technique, *DRRIP* is implemented, which is a state-of-the-art cache replacement policy targeting high-performance [68]. In the experiments, 32 set duelling monitors [108] with $\varepsilon = 1/32$ are used.

6.4 Evaluation

This chapter has evaluated RECAP in terms of performance and energy consumption on a 8-core system with a 16-way and then an 8-way LLC. It also analyses the results



(a) SPEC2006



(b) Parsec

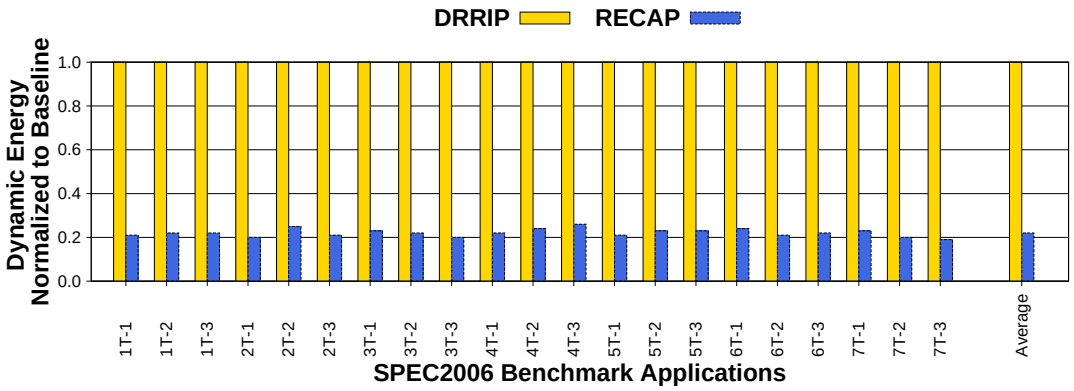
Figure 6.5: Performance of an 8-core system with a 16-way cache.

in terms of access patterns and data category transfer.

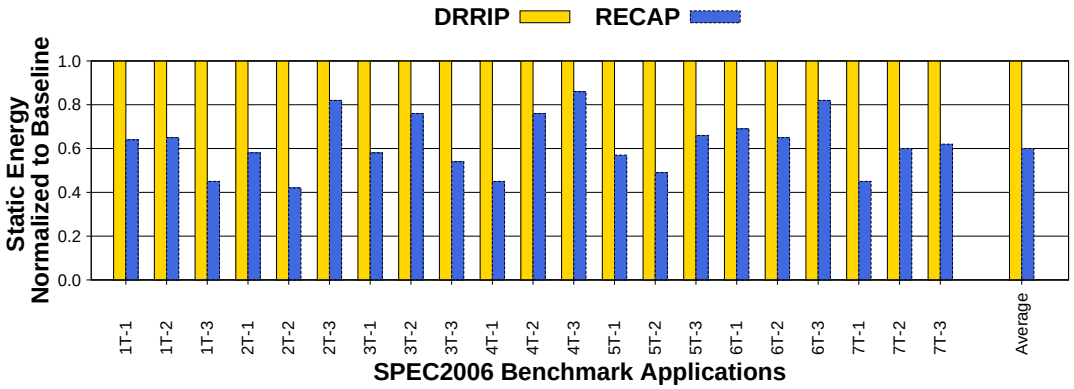
6.4.1 Evaluation on a Sixteen-Way System

Figures 6.5 to 6.7 show the performance and energy consumption for each application and mix. When all cores contend for the cache, DRRIP preserves parts of the working set that lead to better performance. It protects the data of non-thrashing applications from that belonging to the thrashing applications. However, for DRRIP to work effectively, it needs more cache ways than there are cores. In particular, when there are not enough ways to play with, it has difficulty preserving the correct data that benefits overall performance the most. Therefore, it makes suboptimal choices from time-to-time, leading to performance that can be lower than the baseline LRU scheme.

Using RECAP, there is no reduction in performance. In fact, *canneal* achieves a minor performance boost (of 2%) when using RECAP. This is because when the cache is unmanaged, the first three threads use almost all the cache resources, restricting the



(a) Dynamic energy

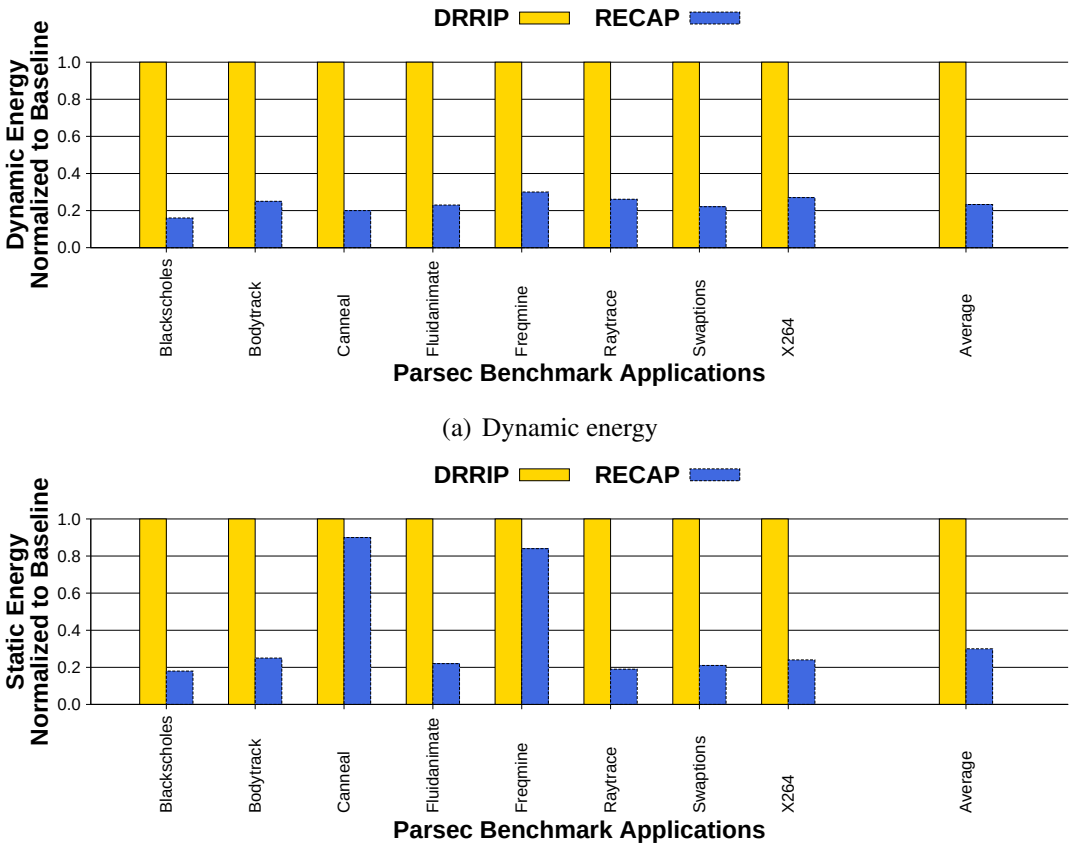


(b) Static energy

Figure 6.6: Energy consumption of an 8-core system with a 16-way cache running SPEC2006 application mixes.

other 5. RECAP efficiently partitions the cache, limiting the number of ways to allocate to these three unfriendly threads, and therefore allowing the others to obtain more performance through an increased share of cache resources. On an average, RECAP achieves 3% better performance compared to DRRIP for the SPEC application mixes and the same performance for the multi-threaded workloads.

RECAP achieves significant dynamic and static energy savings that cannot be realized by DRRIP, since it only targets performance. Figure 6.6 shows the energy consumption of the SPEC application mixes, which averages 22% for dynamic energy and 60% for static energy. Application mix *7T-3* achieves a large reduction of 81% in dynamic energy. The benchmarks in this mix access an average of just 1 or 2 ways, leading to these savings. Notice that, in general, although all application mixes achieve significant energy savings, the mixes on the right achieve more savings than those on the left. This is because the benchmark combinations on the right have a greater number of thrashing applications which are contained in only a couple of ways.



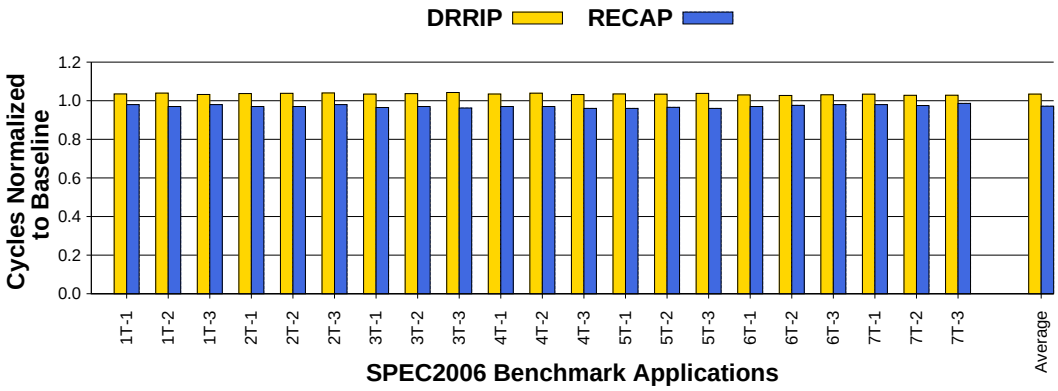
(a) Dynamic energy

(b) Static energy

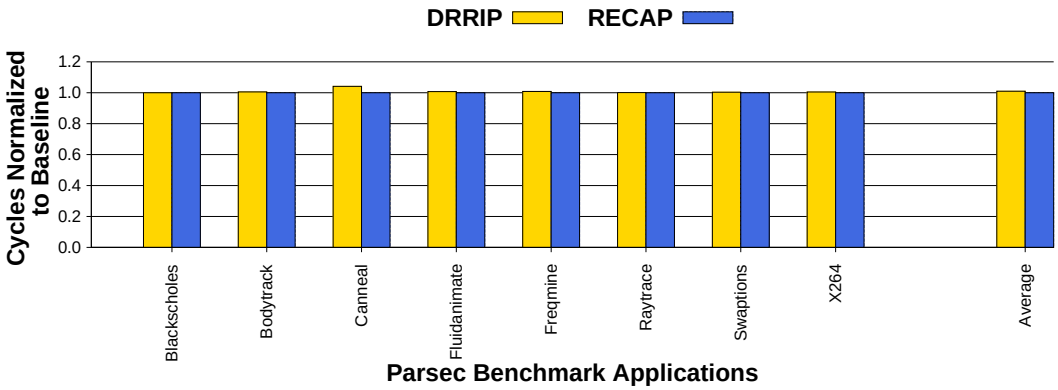
Figure 6.7: Energy consumption of an 8-core system with a 16-way cache running Parsec benchmarks.

In terms of static energy savings, the best energy savings are achieved by groups *2T-2* and *7T-1*. In the former, there are two thrashing applications, whereas there are seven in the latter. RECAP successfully contains these threads in 1 or 2 ways. The other applications in the mixes require only 3 ways in total, meaning that only 4 or 5 ways need to be allocated at any time. The other ways are turned off, leading to the static energy savings of over 50% in each case (42% and 45% static energy consumption compared to the baseline respectively).

The energy consumption of the multi-threaded workloads is shown in Figure 6.7. Here, *blackscholes* achieves the largest reduction in dynamic energy of 88%, again due to the fact that it only accesses 1 or 2 ways, on an average. *canneal* and *freqmine* achieve poor static energy savings. For *canneal*, this is due to all threads needing to access a significant number of ways, meaning that there is little opportunity to turn any off. On the other hand, *freqmine* has one core that needs access to all ways. Even though the other 7 cores can happily survive with fewer ways, this first core would



(a) SPEC2006



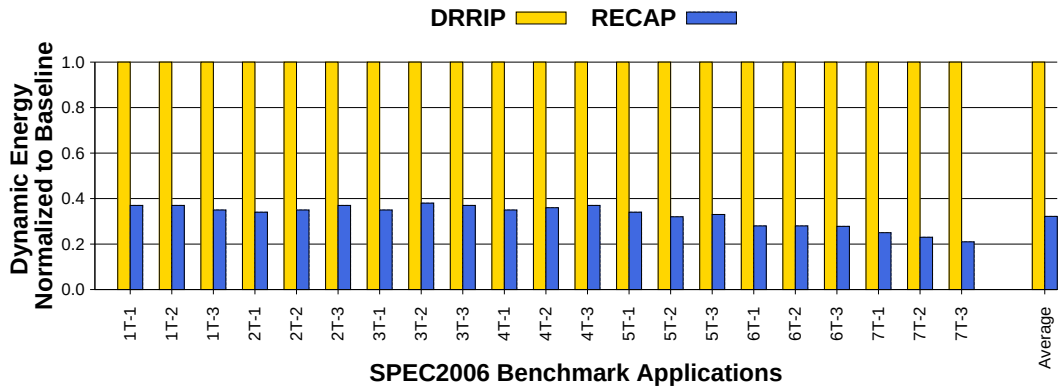
(b) Parsec

Figure 6.8: Performance of an 8-core system with an 8-way cache.

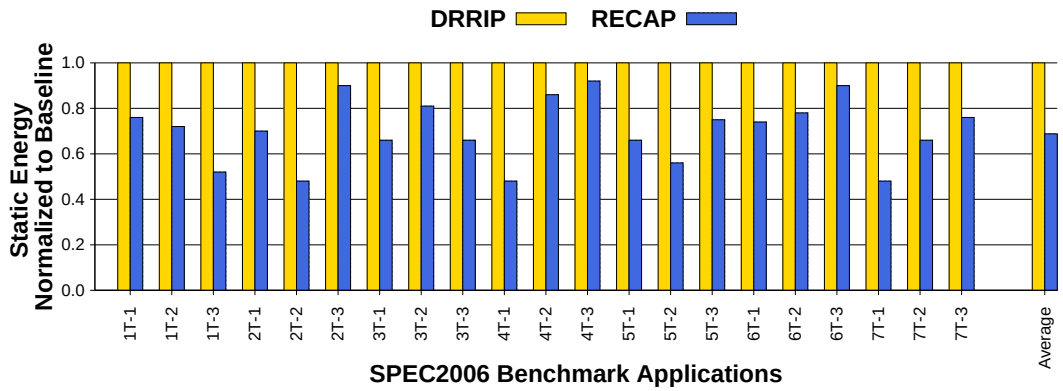
suffer a severe performance loss if it was restricted to fewer ways. Hence, again, for the majority of the execution time, all 16 ways need to be turned on and static energy savings of only 10% can be realized for *canneal*, or 16% for *freqmine*.

6.4.2 Evaluation on an Eight-Way System

Section 6.4.1 presented results when the number of cores was greater than the number of LLC ways. This section evaluates the scalability of RECAP by showing its performance when the number of ways is the same as the number of cores. This is done by using a cache of the same capacity, but half the number of ways (i.e., an 8MB, 8-way LLC). Other cache partitioning schemes cannot cope with this scenario, since they assume that there are more ways than cores within the LLC. Cache partitioning can be implemented in RECAP because it does not enforce a fixed and separate partition to each core, but a pseudo partitioning technique that allows the entire region space to be used by all threads that share the partition. Figure 6.8 shows that this achieves the



(a) Dynamic energy



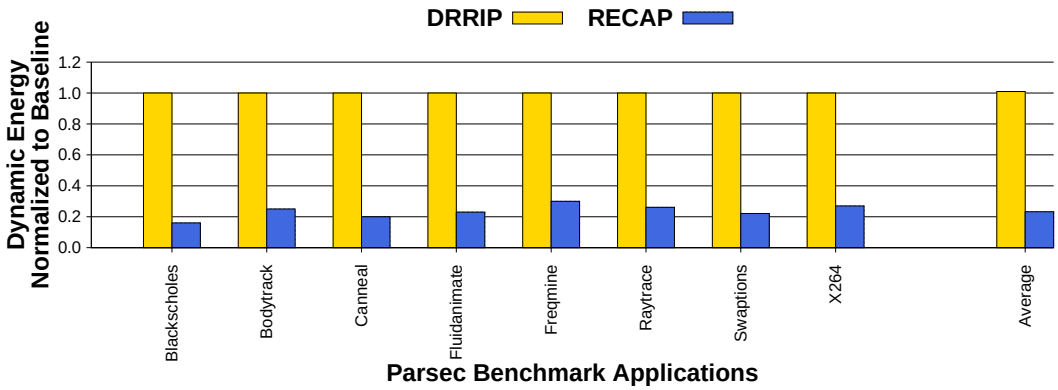
(b) Static energy

Figure 6.9: Energy consumption of an 8-core system with an 8-way cache running SPEC2006 application mixes.

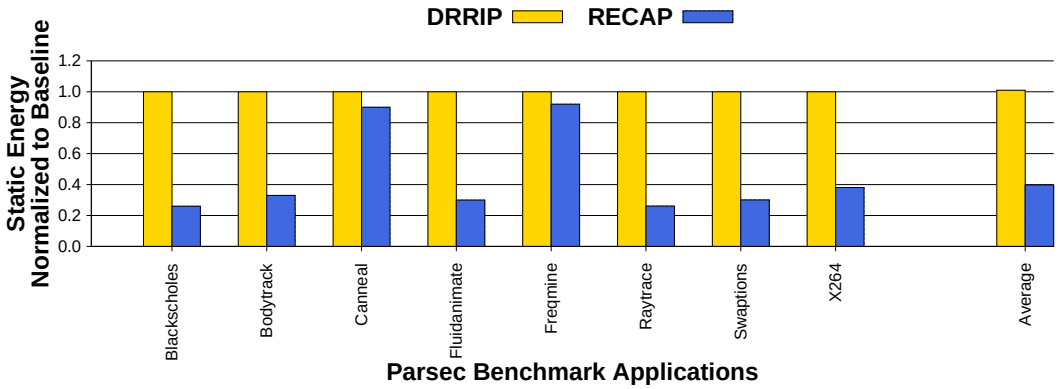
performance of an LRU scheme while Figures 6.9-6.10 show that this can still realize large reductions in energy.

As can be seen in Figure 6.9, the dynamic energy consumption of RECAP is significantly less than the baseline for the SPEC application mixes (32% of the dynamic and 69% of the static energy consumption, on average). The results for the multi-threaded workloads (Figure 6.10) are similar, with an average consumption of 23% dynamic and 40% static energy. These show that even with less scope for energy saving (i.e., 8 ways compared to 16, so more pressure on each way), RECAP is still able to realize significant energy reduction without causing slowdown.

However, groups *2T-3* and *4T-3* achieve disappointing static energy savings of 10% and 8% respectively. Both mixes contain an application that benefits from a large number of cache ways. In *2T-3* it is *xalan*, which requires 7 ways, and in *4T-3* it is *povary*, which requires 6. These applications limit the number of ways that can be turned off, and hence the static energy savings available. This effect can be seen, to an



(a) Dynamic energy



(b) Static energy

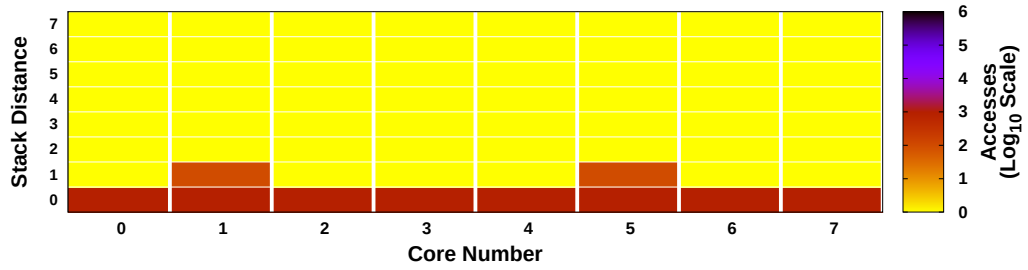
Figure 6.10: Energy consumption of an 8-core system with an 8-way cache running Parsec benchmarks.

extent, in all mixes containing *xalan* (*1T-1*, *1T-2*, *2T-1*, *4T-2*, and *5T-1*).

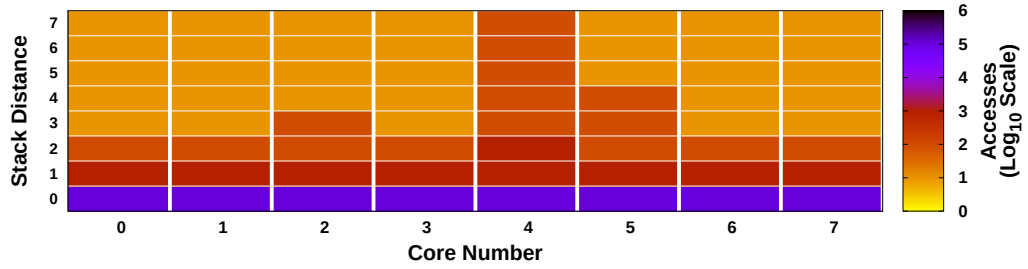
6.4.3 Analysis

Figures 6.11-6.15 analyse these results by showing heat maps corresponding to the stack distance [89] for each core. The darker the colour of a given rectangle, the greater the number of accesses by that core to that distance. In *spec mix*, programs are assigned to ways in the same order as in Table 6.4.

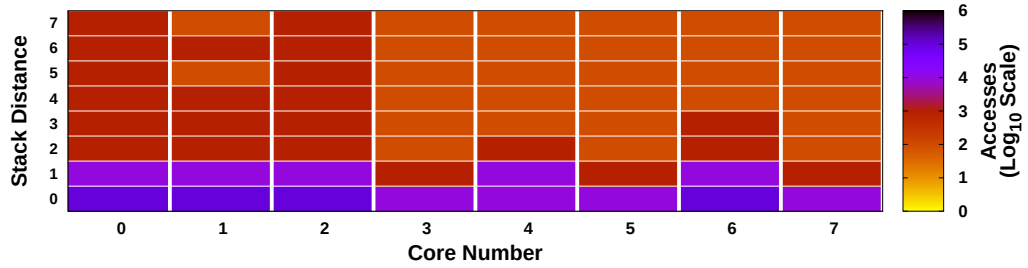
For *blackscholes* it is easy to see that the majority of the accesses are concentrated in the MRU or second position. This means that *blackscholes* can survive with only 2 last-level cache ways, and explains the energy savings shown in Figure 6.10. Turning back to *canneal* and *freqmine*, described previously, it is clear to see that threads 0 - 2 require the most ways for the former, and thread 0 for the latter, which leads to fewer static energy savings.



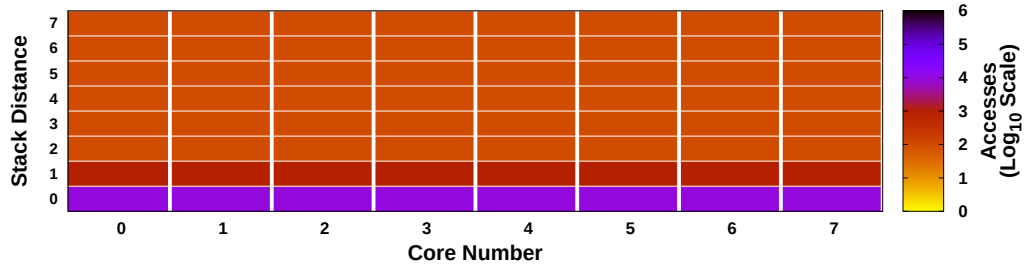
(a) Blackscholes



(b) Bodytrack

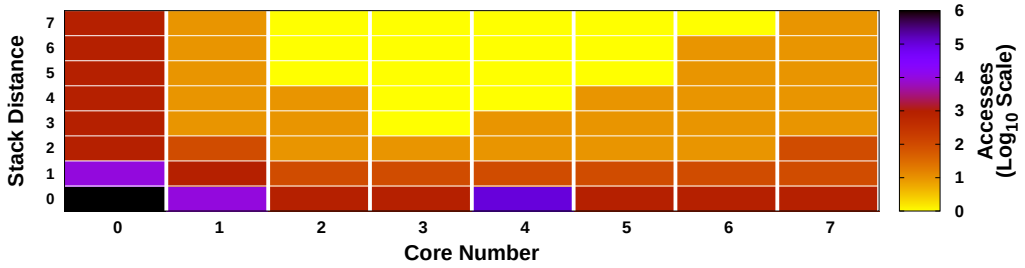


(c) Canneal

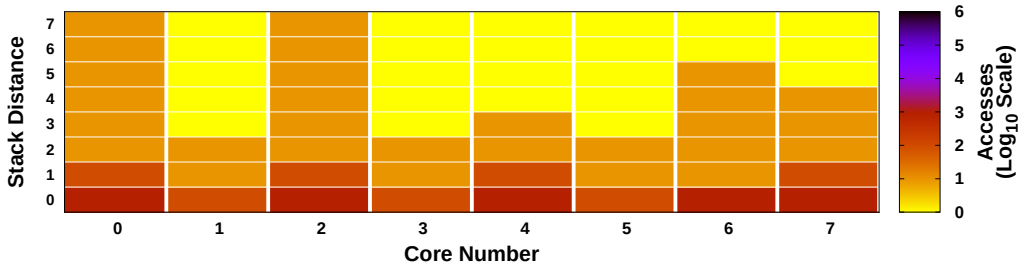


(d) Fluidanimate

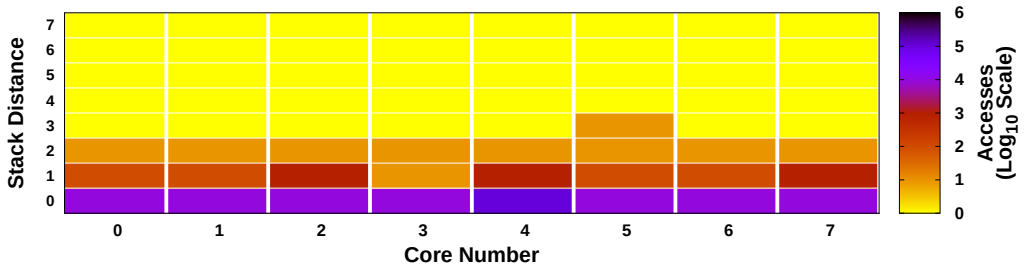
Figure 6.11: Heat maps showing frequency of ways accessed by each core in Parsec applications.



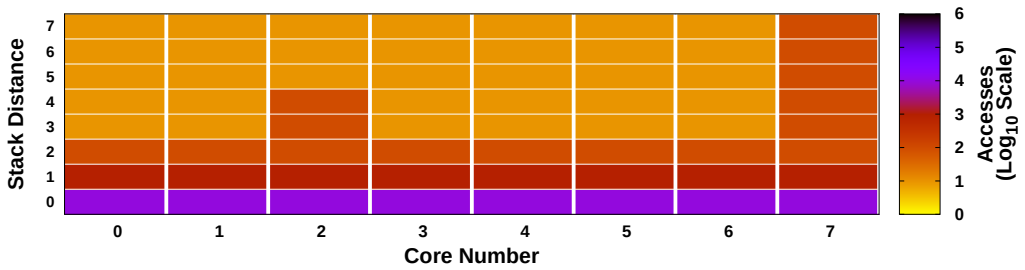
(a) Freqmine



(b) Raytrace

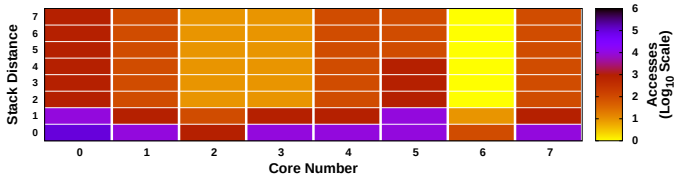


(c) Swaptions

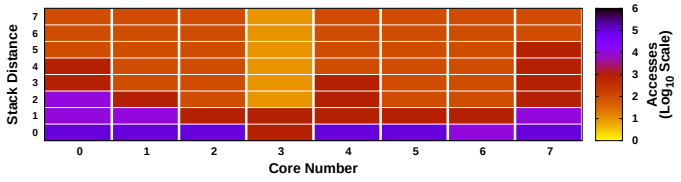


(d) X264

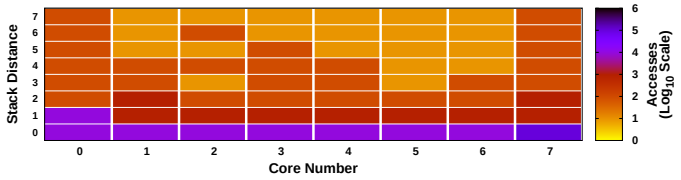
Figure 6.12: Heat maps showing frequency of ways accessed by each core in Parsec applications.



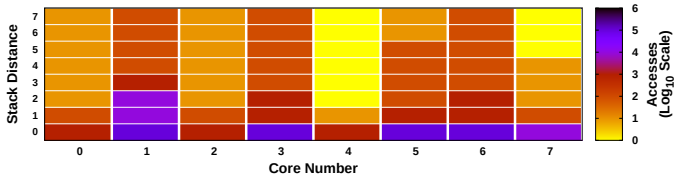
(a) 1T-1



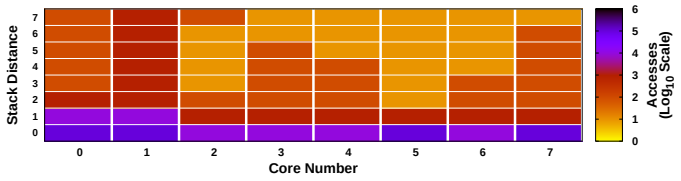
(b) 1T-2



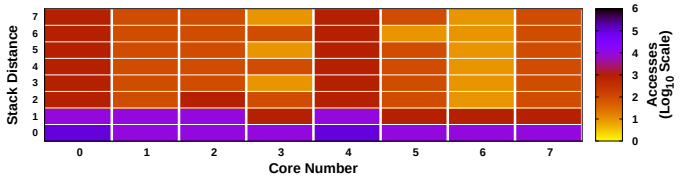
(c) 1T-3



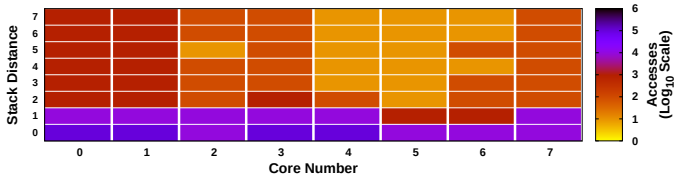
(d) 2T-1



(e) 2T-2

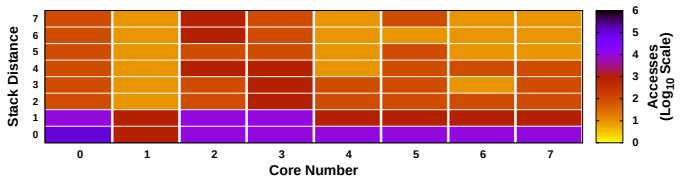


(f) 2T-3

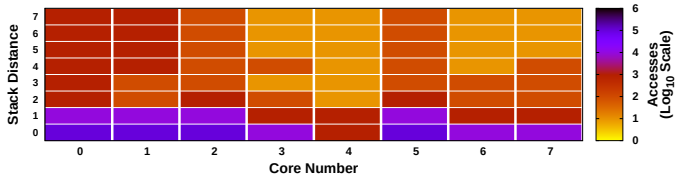


(g) 3T-1

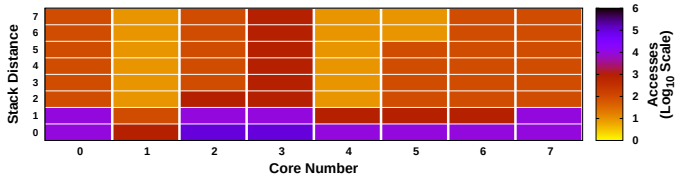
Figure 6.13: Heat maps showing frequency of ways accessed by each core in SPEC2006 application mixes.



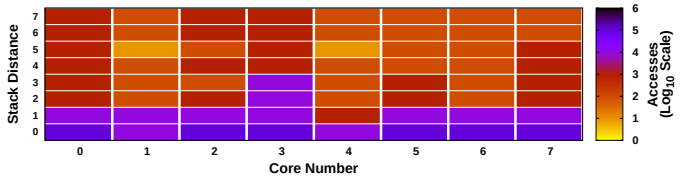
(a) 3T-2



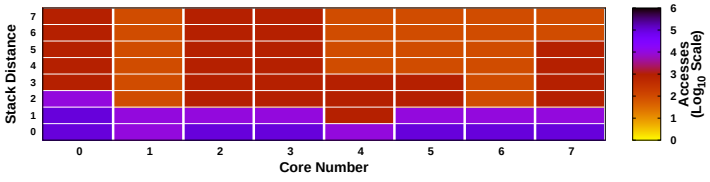
(b) 3T-3



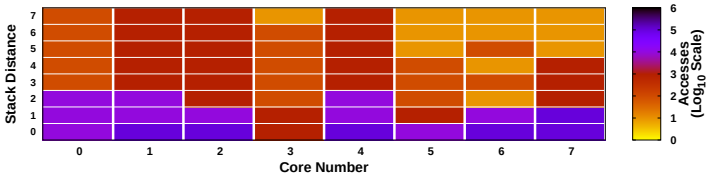
(c) 4T-1



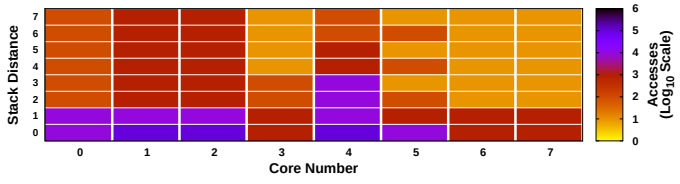
(d) 4T-2



(e) 4T-3

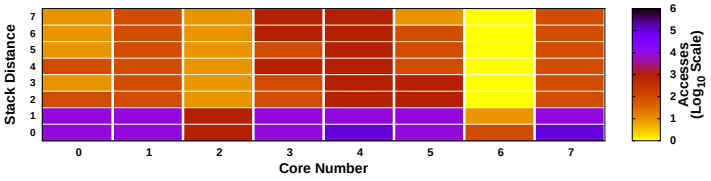


(f) 5T-1

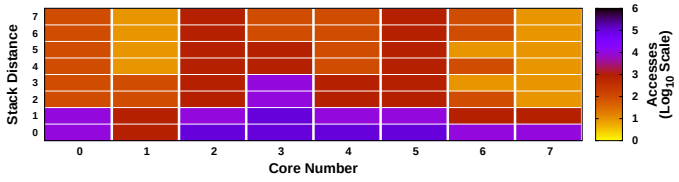


(g) 5T-2

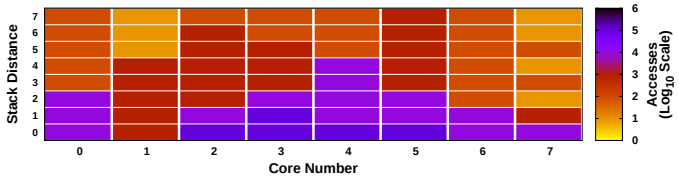
Figure 6.14: Heat maps showing frequency of ways accessed by each core in SPEC2006 application mixes.



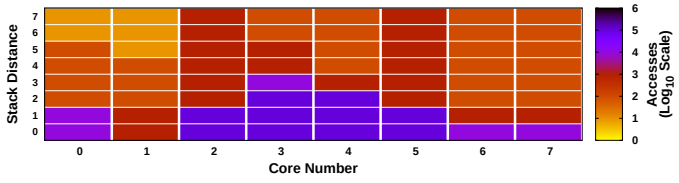
(a) 5T-3



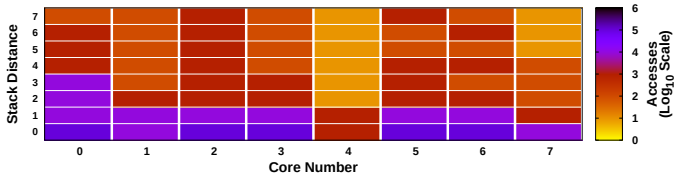
(b) 6T-1



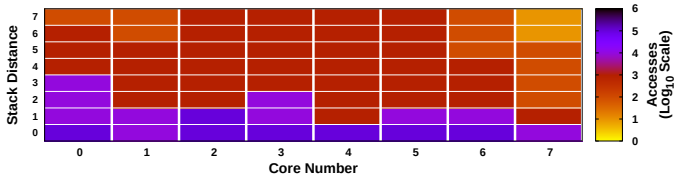
(c) 6T-2



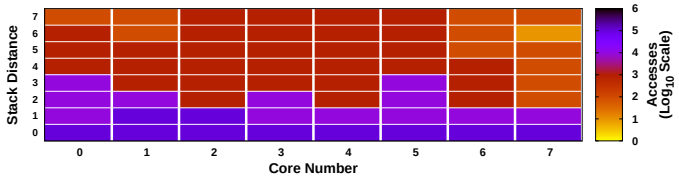
(d) 6T-3



(e) 7T-1



(f) 7T-2



(g) 7T-3

Figure 6.15: Heat maps showing frequency of ways accessed by each core in SPEC2006 application mixes.

For some benchmarks (e.g., *blackscholes*, *fluidanimate*, and *swaptions*), the access patterns are similar across all cores. However, for other applications (e.g., *canneal* and *x264*), there are two distinct groups of threads that have similar access patterns within each group but entirely different patterns across groups. RECAP achieves large dynamic and static energy savings when all cores have similar cache requirements with a small stack distance. However, when the cores have varying cache access patterns, RECAP can realize large dynamic energy savings (by restricting the cores with small stack distance requirements), but is limited in the static energy savings it can achieve (because the cores with large stack distance patterns occupy more ways). In general, the dynamic energy savings achieved by RECAP are influenced by the number of ways allocated to the cores that make the most accesses. Static energy savings, however, are influenced by the cores that require the most amount of cache space.

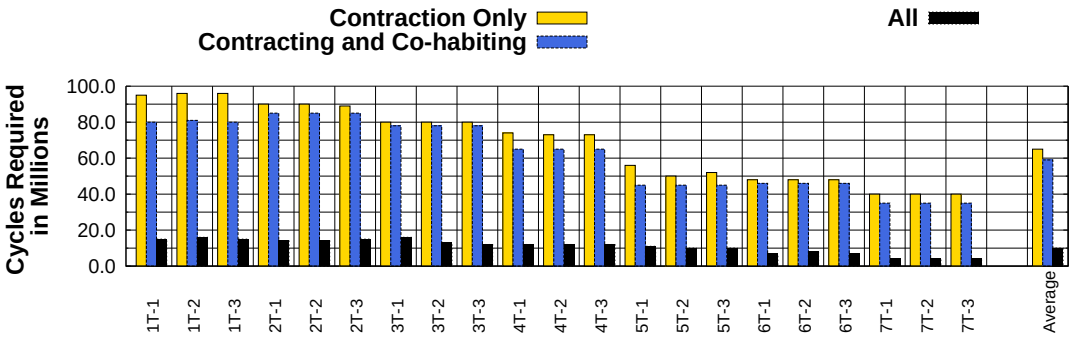
Figures 6.13-6.15 show heatmaps for the SPEC application mixes. Here the colours are darker across the stack distances, meaning that, in general, each core requires more ways to keep high performance. This explains the generally lower energy savings seen in Figure 6.9 compared to Figure 6.10, although since the majority of accesses are made to stack distances 0 and 1, RECAP can still realize significant energy savings.

One application, in particular, makes few accesses to the last-level cache and requires a small stack distance when it does so. This is *dealII*, which is scheduled on core 6 in *1T-1*, core 4 in *2T-1*, and core 6 in *5T-3* and is a significant outlier in this analysis of this set of benchmarks.

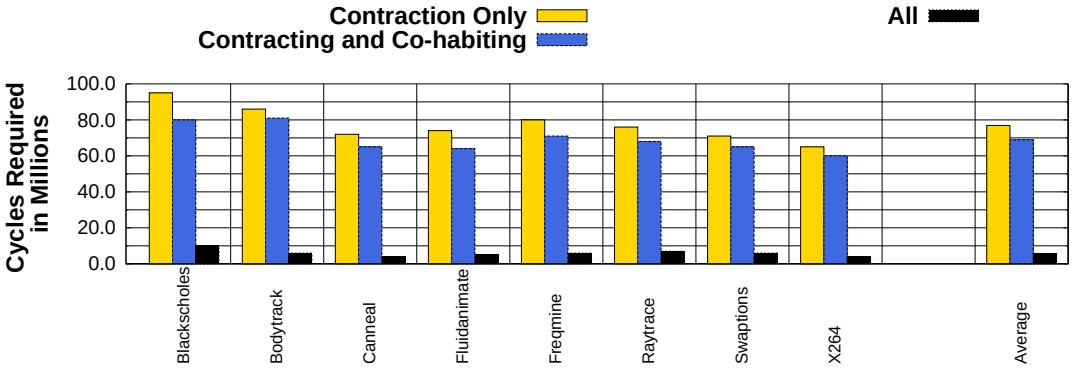
Considering the static energy savings, from Figure 6.9 again, the large savings are achieved by *2T-2* and *7T-1*. In *2T-2*, the six non-thrashing applications (towards the right-hand side of the heatmap) mostly have low stack distance requirements. Hence there is an opportunity to turn off ways. For *7T-1*, the single non-thrashing program on core 7 also mainly accesses stack distances up to five ways, which again allows RECAP to turn parts of the cache off.

6.4.4 Cycles Required for Contraction

Section 6.2.3 described how a core contracts out of ways that it no longer has access to, through the use of the flush bit within the APR. This section now evaluates the effectiveness of this bit by comparing three different approaches to contraction. Figure 6.16 shows the results. In the first scheme, labelled *Contracting Only*, only the contracting core flushes dirty data back to main memory when it leaves a way. The second ap-



(a) SPEC2006



(b) Parsec

Figure 6.16: Number of cycles required to contract out of a way for different flushing schemes.

proach, *Contracting & Cohabiting*, corresponds to the case where data is flushed by the contracting core and any other cores that already have access to that way. Finally, the scheme labelled *All* is the technique used in RECAP where all cores participate in flushing dirty data back main memory.

It is clear from Figure 6.16 that allowing all cores to flush dirty data provides a significant reduction in the time taken to contract out of a way. On an average, for the SPEC application mixes, only 10m cycles are required, compared with 65m for the first scheme and 59m for the second, meaning that it is 85% faster than simply allowing the contracting core to flush data. For Parsec, the corresponding values are 13m, 77m and 69m. In addition, further experiments show that this is also more energy efficient, requiring only 2% of the dynamic energy required when the contracting core or both contracting and co-habituating cores perform the flushing. Although the dynamic energy consumption is larger initially (because additional cores have access to the contracting way), it is for such a small amount of time compared with the two

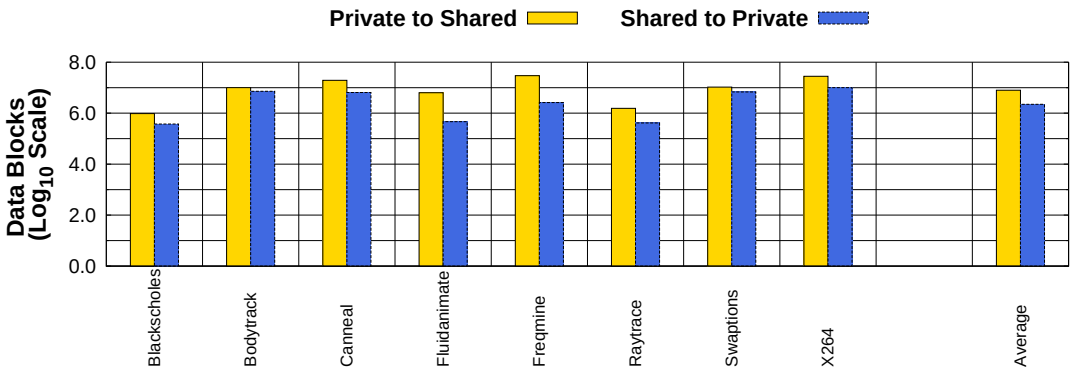


Figure 6.17: Number of data blocks transitioning from private to shared and from shared to private.

other schemes that these overheads are subsumed by the benefits of fast contraction. This clearly shows the benefits of the simple addition of the flush bit to the APR and its effectiveness when contracting out of a way.

6.4.5 Data Category Transitioning

Figure 6.17 shows the number of data blocks that transition category from private to shared or vice-versa. This section shows only Parsec benchmarks because there are no transitions with the SPEC application mixes.

As described in Section 6.2.5, data transitions from shared to private do not affect performance but transitions from private to shared require us to invalidate the line in the private region in LLC, then refetch into the shared region from main memory. The two benchmarks with the smallest number of transitions from private to shared are *blackscholes* and *raytrace*, with 1m and 1.5m transitions respectively. All other benchmarks have 10m transitions, or more (up to 30m for *freqmine*). *blackscholes*, in particular, is known to communicate rarely with other threads in the application, leading to a low number of transitions [99].

On an average, there are 8m transitions from private to shared. However, this is small compared to the overall number of LLC accesses (which is 170m per core on average), so the additional power overheads from refetching data are easily subsumed by the large energy savings.

6.5 Summary

This chapter has proposed RECAP, a region-aware cache partitioning scheme for both multi-programmed and multi-threaded applications running within a shared last-level CMP cache. This approach maintains high performance while saving significant dynamic and static energy. This is achieved by partitioning the cached data into private and shared regions, and only allowing cores to access the ways containing the data type that they seek. Unused ways can be turned off for static energy savings.

This chapter has evaluated RECAP on a 8-core system with a 16-way LLC, showing that it achieves 68% dynamic and 33% static energy savings. Evaluation on the same system with an 8-way LLC maintains high energy savings without any slowdown, showing the scalability of the approach as the number of cores exceeds the number of cache ways.

Chapter 7

Conclusions And Future Work

This chapter presents the summary of contribution of this thesis and discusses possible future directions. The main contribution of this thesis includes the introduction of new energy-efficient cache architectures for single, multi and many core processors.

7.1 Summary Of Contributions

The increasing gap between the speeds of the processor and main memory, motivates the processor vendors to devote the majority of the on-chip transistors to the cache. Traditionally, the same cache design is used in single, multi and many core processors. However, the cache design proposed for a single core processor might not be efficient for the multi core processor and the cache design proposed for the multi core processor might not be efficient for the many core processors. As an example, the reconfigurable caches that are originally proposed for a single core processor, might not be used for a multi core processors. This is due to the frequent phase changes that occur in multi core processors than in single core processor. This dissertation introduces three new cache architectures for three different processor categories that includes single, multi and many core processors.

7.1.1 Single Core Processor

Chapter 4 has presented a new cache architecture, namely the Smart cache for a single core system. Earlier in 2000's, caches were reconfigured during the start of an application and remain in the pre-selected configuration until completion. As the trend shifts towards the run-time reconfiguration, caches are dynamically reconfigured ac-

cording to the program phases of an application. However, the existing state-of-the-art schemes have not changed their cache architecture to support run-time reconfiguration without flushing the data while increasing the associativity and maintaining the same cache size. This is something that the Smart cache avoids during run-time cache reconfiguration.

The Smart cache aligns ways at set boundaries, which is an important feature to avoid flushing of data while increasing the associativity. This alignment of data helps in accessing fewer cache ways for reduced associativity than the state-of-the-art scheme. The state-of-the-art cache scheme is implemented in the simulation infrastructure described in Section 4.4. Thus, Smart cache offers better reduction in dynamic energy compared to the state-of-the-art scheme. On an average, the energy-delay of Smart cache was 18% better than the state-of-the-art schemes.

SPEC CPU2000 and SPEC CPU2006 benchmarks were used as workloads. Machine learning was used to predict the required cache size and associativity for the current program phase, based on the observation of the previous program phase. When all three caches, namely instruction, data and level-2 cache were reconfigured simultaneously, Smart cache achieves an energy-delay reduction of 50% compared to the baseline. Results were presented for both, decision tree and linear regression models. Decision tree is preferred over linear regression, due to reduced hardware overhead associated with the decision tree. Leave-one-out cross validation was used for evaluation. The superiority of the decision tree model was shown by training the model with SPEC CPU2000 benchmarks and predicting SPEC CPU2006 benchmarks.

The Smart cache architecture was extended to multi cores with two and four cores. A decision tree model was used to predict the required cache size and associativity. On an average, Smart cache achieved an energy-delay reduction of 45% and 12% for two and four core workloads respectively, when compared to a baseline configuration described in Section 4.4

The Smart cache is most suitable for embedded systems that have a single processing core and are expected to be power efficient. This can range from mobile phones to hand-held tablets.

7.1.2 Multi Core Processor

The Smart cache was good for a single core system. Extending the Smart cache to multi cores, provides a diminishing reduction in energy-delay. This diminishing reduction in

energy-delay was observed from the results of two core and four core systems of the Smart cache. This is due to the frequent phase changes that occurs among the applications that run on different cores that share a LLC. This results in frequent flushing of data that hurts the performance. This motivates the need for a new cache architecture for multi core system that share the LLC.

When the cores share the LLC (a core-to-LLC way ratio of 1:4 is used), they compete with each other for the LLC space. This leads to a behaviour where one application running on one core tries to evict another application's data that runs on a different core. To overcome this behaviour, cache partitioning was used. The state-of-the-art cache partitioning schemes were targeted purely for high-performance. Chapter 5 has presented a new cache partitioning scheme, namely cooperative partitioning that offers significant reduction in energy (both static and dynamic energy), while maintaining high-performance.

SPEC CPU2006 benchmarks were used as workloads. Cooperative partitioning monitors each individual applications' cache requirements through the shadow tag, which adds less than 2% hardware overhead to the LLC. Basically, a shadow tag was used to compute the stack distance and this information was computed for each application running on each individual core. Later, they were passed to the partitioning algorithm that makes the partition decision based on the benefit ratio or percentage improvement in the hit ratio, compared to its current allocations. Cache ways were allocated to an application with more improvement in hit ratio, when compared to other applications. A threshold value of 5% was used in the partitioning algorithm, any improvement in hit ratio of the competing applications above this value gets the cache way and any improvement in hit ratio below this value does not get a cache way. The unallocated cache ways were turned-off to reduce the static energy.

Cooperative partitioning maintains a way-aligned data, meaning that each core knows the cache ways that has its data. This way-alignment helps in the reduction of dynamic energy, as each core accesses only the required cache ways that has its data. The state-of-the-art cache partitioning along with fixed partitioning is implemented in the simulation infrastructure described in Section 5.3. Results show that on an average, cooperative partitioning achieves 67% and 25% reduction in dynamic and static energy, respectively, for a two core system and achieves 83% and 20% reduction in dynamic and static energy, respectively, for a four core system, compared to the baseline configuration.

The assumption of this core-to-LLC way ratio holds for desktop systems with two

or four cores, sharing an 8-way or a 16-way LLC respectively. Commercial examples that have this core-to-LLC way ratio include Intel's two-core Core2Duo and Intel's four-core i7.

7.1.3 Many Core Processors

Results from cooperative partitioning have shown that this scheme is scalable. This was inferred from the reduction in dynamic and static energy for two to four cores, either the reduction numbers stay the same or it improves. Cooperative partitioning was good for systems that have more LLC ways compared to cores, with a core-to-LLC ratio of 1:4.

However, with the increasing number of cores, this ratio is no longer valid. When the core-to-LLC ratio drops to 1:2 or 1:1, the state-of-the-art partitioning schemes and cooperative partitioning cannot be used. This motivates the need for a new partitioning scheme targeting many cores. Chapter 6 has presented a new cache partitioning scheme, namely region aware cache partitioning (RECAP) that offers a significant reduction in energy (both static and dynamic energy), while maintaining high-performance.

Like cooperative partitioning, RECAP monitors individual application's cache requirement through the shadow tag and the stack distance calculated from the shadow tag for each individual core, is passed to the partitioning algorithm. This partitioning algorithm differs from cooperative partitioning in Five aspects.

First, RECAP allocates LLC ways to all applications that show 5% improvement in hit ratio. Whereas, in cooperative partitioning, the application that shows more improvement in hit ratio compared to other applications and also this improvement should be at least 5% more than its current way allocations, will get the LLC way.

Second, after allocating the cache ways, RECAP groups the applications that has similar cache requirement in to one group and if no similarity was observed, it creates a new group.

Third, RECAP left justifies the groupings. Therefore, Way 0 will be shared by all the cores, Way 1 to remaining will be shared by cores depending on the cache requirement of the grouping.

Four, RECAP uses the directory information to distinguish between shared and private data. As mentioned above, private data was left justified and shared data was right justified.

Fifth, RECAP proposes a pseudo-way aligned data, meaning that a LLC way may contain other cores data as well. Whereas in cooperative partitioning, a LLC way has only one core data.

SPEC CPU2006 and PARSEC 2.1 benchmarks each were used as workloads for multi-programmed and multi-threaded scenarios. The core consult only the required ways that have its data. When the request is for a shared data, then only the way that has the shared data is activated. This offers a significant reduction in dynamic energy. The unused cache ways can be turned off to reduce the static power. The state-of-the-art cache replacement policy is implemented in the simulation infrastructure described in Section 6.3. Results show that on an average, for an eight core system, RECAP offers 66% and 33% reduction in dynamic and static energy and 77% and 60% reduction in dynamic and static energy for multi-programmed and multi-threaded workloads respectively, compared to the baseline configuration.

The assumption of this core-to-LLC way ratio holds for server systems with 8 or 16, sharing an 8-way or 16-way LLC respectively. For such systems, RECAP provides an energy-efficient cache architecture whilst maintaining high-performance.

7.2 Future Work

This thesis has proposed new cache architectures for single, multi and many core processors, targeting their application in embedded, desktop and server computing systems respectively. The following describes the future directions where the previously gained knowledge about caches can be applied.

7.2.1 Heterogeneous Systems

High-performance computing systems create an opportunity for the usage of accelerators to speed-up the critical regions of the program execution. When such accelerators are used the system is no longer homogeneous, it becomes heterogeneous system. Different cores operating at different frequencies with different ISAs help in accelerating the speed of execution.

In such a system, managing the LLC according to the heterogeneous core's requirement will be challenging. A commercial example is an AMD-Fusion core that has an on-chip Central Processing Unit (CPU) and Graphics Processing Unit (GPU) that share the LLC. In a homogeneous CPU sharing the LLC, the LLC is accessed less

frequently compared to the level 1 cache. As most of the requests are served by the level 1 cache.

However, in a heterogeneous system, consisting of CPU and GPU, GPUs access frequency of the LLC is much more compared to the CPU. In such a system, the proposed cooperative partitioning and RECAP will reduce a huge amount of dynamic energy, as the GPUs LLC access frequency is more. However, maintaining a way-aligned data in the presence of CPU and GPU will be an interesting area to explore.

7.2.2 Many Core CMPs

As the number of processing cores increases, for example, in a 1000 core system, on-chip caches are banked and are shared among several cores through the network. In such systems, network delay is a major bottleneck for improving the performance. A new cache architecture that keeps the data close to the processing core should be proposed.

State-of-the-art spill/receive architectures and Reactive-NUCA schemes tries to keep the data close to the processing core. These schemes keep the currently required data close to the requesting core. However a lookahead pre-fetching of the data and keeping the future data close to the processing core will overlap the network delay and will allow better utilization of cache that results in improved performance.

7.2.3 Graphics Processing Unit

Another direction of future research is extending the RECAP cache partitioning scheme to GPUs. GPUs have a massive number of processing cores. Nvidia Fermi architecture with 32 cores and on-chip cache hierarchy is one such example.

Due to the increasing demand for high-performance computation, the processor vendors allow the inclusion of cache hierarchies to offer better performance. The interference caused in co-executing warps is avoided by effectively partitioning the shared cache according to the warp requirement thereby improving overall performance and also reduces dynamic energy in shared cache by accessing the corresponding warp's region upon every cache access.

RECAP flushes the data present in an LLC way to make it way-aligned. However in the presence of GPUs, this flushing incurs more performance loss. Allowing a gradual flushing of the data to maintain a way-aligned LLC will be an interesting area to explore.

7.2.4 Using Compiler Hints

Finally, compilers get the opportunity to see the source code of the program before it is actually executed in the processor. Using compilers to give hints about the upcoming program phases of an application, improves the cache reconfiguration decision in a single core processor or cache partitioning decision in a multi or many core processors.

Frequent cache reconfiguration results in poor performance. The state-of-the-art cache reconfiguration or cache partitioning schemes detect the phase on-line and reconfigure the cache accordingly. However, the existing schemes does not know the criticality of the phase that has been detected.

Compilers have this information through compiling the source code before its been executed. Passing the information from compilers to the run-time systems, help in improving the performance of the system by making the reconfiguration or partition decision only when required. This leads to a benefit based cache reconfiguration or cache partitioning in single core and multi core processors respectively. This will form a backbone to all the existing cache reconfiguration or cache partitioning schemes.

Bibliography

[1] Jaume Abella and Antonio González. On reducing register pressure and energy in multiple-banked register files. In *Proceedings of the 21st International Conference on Computer Design*, ICCD '03, pages 14–, Washington, DC, USA, 2003. IEEE Computer Society.

[2] Jaume Abella and Antonio González. Power efficient data cache designs. In *Proceedings of the 21st International Conference on Computer Design*, ICCD '03, pages 8–, Washington, DC, USA, 2003. IEEE Computer Society.

[3] Jaume Abella and Antonio González. Heterogeneous way-size cache. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 239–248, New York, NY, USA, 2006. ACM.

[4] Amit Agarwal, Hai Li, and Kaushik Roy. Drg-cache: a data retention gated-ground cache for low power. In *Proceedings of the 39th annual Design Automation Conference*, DAC '02, pages 473–478, New York, NY, USA, 2002. ACM.

[5] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst.*, 6(4):393–431, November 1988.

[6] Anant Agarwal and Stephen D. Pudar. Column-associative caches: a technique for reducing the miss rate of direct-mapped caches. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 179–190, New York, NY, USA, 1993. ACM.

[7] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proceedings of the 42nd annual Southeast regional conference*, ACM-SE 42, pages 267–272, New York, NY, USA, 2004. ACM.

[8] David H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 32, pages 248–259, Washington, DC, USA, 1999. IEEE Computer Society.

[9] D.H. Albonesi. Dynamic ipc/clock rate optimization. In *Computer Architecture, 1998. Proceedings. The 25th Annual International Symposium on*, pages 282 – 292, jun-1 jul 1998.

[10] B. Batson and T.N. Vijaykumar. Reactive-associative caches. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pages 49 –60, 2001.

[11] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, June 1966.

[12] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[13] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, May 2011.

[14] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 318–329, Washington, DC, USA, 2008. IEEE Computer Society.

[15] Alper Buyuktosunoglu, David Albonesi, Stanley Schuster, David Brooks, Pradip Bose, and Peter Cook. A circuit level implementation of an adaptive issue queue for power-aware microprocessors. In *Proceedings of the 11th Great Lakes symposium on VLSI*, GLSVLSI '01, pages 73–78, New York, NY, USA, 2001. ACM.

[16] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer*

Architecture, HPCA '96, pages 244–, Washington, DC, USA, 1996. IEEE Computer Society.

[17] Francisco J. Cazorla, Peter M.W. Knijnenburg, Rizos Sakellariou, Enrique Fernández, Alex Ramirez, and Mateo Valero. Predictable performance in SMT processors. In *Proceedings of the 1st conference on Computing frontiers*, CF '04, pages 433–443, New York, NY, USA, 2004. ACM.

[18] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[19] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 264–276, Washington, DC, USA, 2006. IEEE Computer Society.

[20] Jichuan Chang and Gurindar S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, pages 242–252, New York, NY, USA, 2007. ACM.

[21] Liming Chen, Xuecheng Zou, Jianming Lei, and Zhenglin Liu. Dynamically reconfigurable cache for low-power embedded system. In *Third International Conference on Natural Computation, 2007. ICNC 2007*, volume 5, pages 180 –184, Aug. 2007.

[22] William Y. Chen, Roger A. Bringmann, Scott A. Mahlke, Richard E. Hank, and James E. Sicolo. An efficient architecture for loop based data preloading. In *Proceedings of the 25th annual international symposium on Microarchitecture*, MICRO 25, pages 92–101, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[23] Derek Chiou, Srinivas Devadas, Larry Rudolph, and Boon S. Ang. Dynamic cache partitioning via columnization. In *In Proceedings of Design Automation Conference*, 2000.

[24] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 55–, Washington, DC, USA, 2003. IEEE Computer Society.

[25] Seungryul Choi and Donald Yeung. Learning-based SMT processor resource distribution via hill-climbing. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 239–251, Washington, DC, USA, 2006. IEEE Computer Society.

[26] Robert Neale Cooksey. *Content-sensitive data prefetching*. PhD thesis, Boulder, CO, USA, 2002. AAI3043511.

[27] Blas A. Cuesta, Alberto Ros, María E. Gómez, Antonio Robles, and José F. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 93–104, New York, NY, USA, 2011. ACM.

[28] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideovt, Ernest Bassous, and Andre R. Leblanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits Newsletter, IEEE*, 12(1):38 – 50, winter 2007.

[29] Pedro Diaz and Marcelo Cintra. Stream chaining: exploiting multiple levels of correlation in data prefetching. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 81–92, New York, NY, USA, 2009. ACM.

[30] Steve Dropsho, Alper Buyuktosunoglu, Rajeev Balasubramonian, David H. Albonesi, Sandhya Dwarkadas, Greg Semeraro, Grigorios Magklis, and Michael L. Scott. Integrating adaptive on-chip storage structures for reduced dynamic power. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 141–, Washington, DC, USA, 2002. IEEE Computer Society.

[31] Christophe Dubach, Timothy Jones, and Michael O'Boyle. Microarchitectural design space exploration using an architecture-centric approach. In *Proceedings*

of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 40, pages 262–271, Washington, DC, USA, 2007. IEEE Computer Society.

[32] Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, and Michael F. P. O'Boyle. A predictive model for dynamic microarchitectural adaptivity control. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 485–496, Washington, DC, USA, 2010. IEEE Computer Society.

[33] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 335–346, New York, NY, USA, 2010. ACM.

[34] Aristides Efthymiou and Jim D. Garside. Adaptive pipeline structures fo speculation control. In *Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems*, ASYNC '03, pages 46–, Washington, DC, USA, 2003. IEEE Computer Society.

[35] Brian A. Fields, Rastislav Bodík, Mark D. Hill, and Chris J. Newburn. Using interaction costs for microarchitectural bottleneck analysis. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 228–, Washington, DC, USA, 2003. IEEE Computer Society.

[36] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 148–157, Washington, DC, USA, 2002. IEEE Computer Society.

[37] Daniele Folegnani and Antonio González. Energy-effective issue logic. In *Proceedings of the 28th annual international symposium on Computer architecture*, ISCA '01, pages 230–239, New York, NY, USA, 2001. ACM.

[38] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of the 25th annual international symposium*

on Microarchitecture, MICRO 25, pages 102–110, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[39] Mrinmoy Ghosh, Emre Ozer, Simon Ford, Stuart Biles, and Hsien-Hsin S. Lee. Way guard: a segmented counting bloom filter approach to reducing energy for set-associative caches. In *Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design*, ISLPED '09, pages 165–170, New York, NY, USA, 2009. ACM.

[40] Antonio González, Carlos Aliagas, and Mateo Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Proceedings of the 9th international conference on Supercomputing*, ICS '95, pages 338–347, New York, NY, USA, 1995. ACM.

[41] Ann Gordon-Ross, Jeremy Lau, and Brad Calder. Phase-based cache reconfiguration for a highly-configurable two-level cache hierarchy. In *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, GLSVLSI '08, pages 379–382, New York, NY, USA, 2008. ACM.

[42] Ann Gordon-Ross and Frank Vahid. A self-tuning configurable cache. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 234–237, New York, NY, USA, 2007. ACM.

[43] Ann Gordon-Ross, Frank Vahid, and Nikil Dutt. Fast configurable-cache tuning with a unified second-level cache. In *Proceedings of the 2005 international symposium on Low power electronics and design*, ISLPED '05, pages 323–326, New York, NY, USA, 2005. ACM.

[44] Fei Guo, Yan Solihin, Li Zhao, and Ravishankar Iyer. A framework for providing quality of service in chip multi-processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 343–355, Washington, DC, USA, 2007. IEEE Computer Society.

[45] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[46] Erik G. Hallnor and Steven K. Reinhardt. A fully associative software-managed cache design. In *Proceedings of the 27th annual international symposium on*

Computer architecture, ISCA '00, pages 107–116, New York, NY, USA, 2000. ACM.

[47] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 184–195, New York, NY, USA, 2009. ACM.

[48] Atsushi Hasegawa, Ikuya Kawasaki, Kouji Yamada, Shinichi Yoshioka, Shumpei Kawasaki, and Prasenjit Biswas. Sh3: High code density, low power. *IEEE Micro*, 15(6):11–19, December 1995.

[49] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[50] Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Don Newell, Vineet Chadha, and Jaideep Moses. Rate-based QoS techniques for cache/memory in cmp platforms. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 479–488, New York, NY, USA, 2009. ACM.

[51] Enric Herrero, José González, and Ramon Canal. Distributed cooperative caching. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 134–143, New York, NY, USA, 2008. ACM.

[52] Enric Herrero, José González, and Ramon Canal. Elastic cooperative caching: an autonomous dynamically adaptive memory hierarchy for chip multiprocessors. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 419–428, New York, NY, USA, 2010. ACM.

[53] Mark Donald Hill. *Aspects of cache memory and instruction buffer performance*. PhD thesis, 1987. AAI8813907.

[54] Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 38–48, New York, NY, USA, 2003. ACM.

[55] Lisa R. Hsu, Steven K. Reinhardt, Ravishankar Iyer, and Srihari Makineni. Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 13–22, New York, NY, USA, 2006. ACM.

[56] Michael Huang, Jose Renau, Seung-Moon Yoo, and Josep Torrellas. L1 data cache decomposition for energy efficiency. In *Proceedings of the 2001 international symposium on Low power electronics and design*, ISLPED '01, pages 10–15, New York, NY, USA, 2001. ACM.

[57] Christopher J. Hughes, Jayanth Srinivasan, and Sarita V. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, pages 250–261, Washington, DC, USA, 2001. IEEE Computer Society.

[58] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *Proceedings of the 1999 international symposium on Low power electronics and design*, ISLPED '99, pages 273–275, New York, NY, USA, 1999. ACM.

[59] Engin Ipek, Bronis R. de Supinski, Martin Schulz, and Sally A. McKee. An approach to performance prediction for parallel applications. In *Proceedings of the 11th international Euro-Par conference on Parallel Processing*, Euro-Par'05, pages 196–205, Berlin, Heidelberg, 2005. Springer-Verlag.

[60] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 186–197, New York, NY, USA, 2007. ACM.

[61] Engin İpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 195–206, New York, NY, USA, 2006. ACM.

[62] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 359–370, Washington, DC, USA, 2006. IEEE Computer Society.

[63] Ravi Iyer. CQoS: a framework for enabling qos in shared caches of cmp platforms. In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, pages 257–266, New York, NY, USA, 2004. ACM.

[64] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. QoS policies and architecture for cache/memory in cmp platforms. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '07, pages 25–36, New York, NY, USA, 2007. ACM.

[65] Magnus Jahre and Lasse Natvig. A light-weight fairness mechanism for chip multiprocessor memory systems. In *Proceedings of the 6th ACM conference on Computing frontiers*, CF '09, pages 1–10, New York, NY, USA, 2009. ACM.

[66] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 208–219, New York, NY, USA, 2008. ACM.

[67] Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely, and Joel Emer. Cruise: cache replacement and utility-aware scheduling. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 249–260, New York, NY, USA, 2012. ACM.

[68] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM.

[69] P. J. Joseph, Kapil Vaswani, and Matthew J. Thazhuthaveetil. Construction and use of linear regression models for processor performance analysis. In *Pro-*

ceedings of the 12th International Symposium on High-Performance Computer Architecture, HPCA '06, Austin, TX, USA, 2006. IEEE Computer Society.

[70] P. J. Joseph, Kapil Vaswani, and Matthew J. Thazhuthaveetil. A predictive performance model for superscalar processors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 161–170, Washington, DC, USA, 2006. IEEE Computer Society.

[71] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 364–373, New York, NY, USA, 1990. ACM.

[72] Toni Juan, Tomás Lang, and Juan J. Navarro. The difference-bit cache. In *Proceedings of the 23rd annual international symposium on Computer architecture*, ISCA '96, pages 114–120, New York, NY, USA, 1996. ACM.

[73] Tejas S. Karkhanis and James E. Smith. A first-order superscalar processor model. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 338–, Washington, DC, USA, 2004. IEEE Computer Society.

[74] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th annual international symposium on Computer architecture*, ISCA '01, pages 240–251, New York, NY, USA, 2001. ACM.

[75] Kamil Kedzierski, Francisco J. Cazorla, Roberto Gioiosa, Alper Buyuktosunoglu, and Mateo Valero. Power and performance aware reconfigurable cache for cmps. In *Proceedings of the Second International Forum on Next-Generation Multicore/Manycore Technologies*, IFMT '10, pages 1:1–1:12, New York, NY, USA, 2010. ACM.

[76] R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill. Inexpensive implementations of set-associativity. In *Proceedings of the 16th annual international symposium on Computer architecture*, ISCA '89, pages 131–139, New York, NY, USA, 1989. ACM.

[77] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 211–222, New York, NY, USA, 2002. ACM.

[78] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society.

[79] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. The filter cache: an energy efficient memory structure. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, pages 184–193, Washington, DC, USA, 1997. IEEE Computer Society.

[80] Vasileios Kontorinis, Amirali Shayan, Dean M. Tullsen, and Rakesh Kumar. Reducing peak power with a table-driven adaptive processor core. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 189–200, New York, NY, USA, 2009. ACM.

[81] Benjamin C. Lee and David Brooks. Efficiency trends and limits from comprehensive microarchitectural adaptivity. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 36–47, New York, NY, USA, 2008. ACM.

[82] Benjamin C. Lee and David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 185–194, New York, NY, USA, 2006. ACM.

[83] Benjamin C. Lee and David M. Brooks. Illustrative design space studies with microarchitectural regression models. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 340–351, Washington, DC, USA, 2007. IEEE Computer Society.

[84] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 249–258, New York, NY, USA, 2007. ACM.

[85] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Comput.*, 50(12):1352–1361, December 2001.

[86] Jaekyu Lee and Hyesoon Kim. Tap: A tlp-aware cache management policy for a cpu-gpu heterogeneous architecture. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1 –12, feb. 2012.

[87] Yingmin Li, B. Lee, D. Brooks, Zhigang Hu, and K. Skadron. Cmp design space exploration subject to physical constraints. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 17 – 28, feb. 2006.

[88] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: speculation control for energy reduction. In *Proceedings of the 25th annual international symposium on Computer architecture*, ISCA '98, pages 132–141, Washington, DC, USA, 1998. IEEE Computer Society.

[89] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970.

[90] Yan Meng, Timothy Sherwood, and Ryan Kastner. On the limits of leakage power reduction in caches. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 154–165, Washington, DC, USA, 2005. IEEE Computer Society.

[91] J. Merino, V. Puente, and J.A. Gregorio. ESP-NUCA: A low-cost adaptive non-uniform cache architecture. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –10, jan. 2010.

[92] Albert Ma Michael, Michael Zhang, and Krste Asanovic. Way memoization to reduce fetch energy in instruction caches. In *ISCA Workshop on Complexity Effective Design*. MIT, 2001.

[93] A. M. Molnos, S. D. Cotofana, M. J. M. Heijligers, and J. T. J. van Eijndhoven. Static cache partitioning robustness analysis for embedded on-chip multi-processors. In *Proceedings of the 3rd conference on Computing frontiers*, CF '06, pages 353–360, New York, NY, USA, 2006. ACM.

[94] A. M. Molnos, M. J. M. Heijligers, S. D. Cotofana, and J. T. J. van Eijndhoven. Compositional, efficient caches for a chip multi-processor. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, DATE '06, pages 345–350, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[95] Anca M. Molnos, Sorin D. Cotofana, Marc J.M. Heijligers, and Jos T.J. van Eijndhoven. Throughput optimization via cache partitioning for embedded multi-processors. In *Embedded Computer Systems: Architectures, Modeling and Simulation, 2006. IC-SAMOS 2006. International Conference on*, pages 185 –191, july 2006.

[96] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82 –85, jan. 1998.

[97] S.P. Muralidhara, M. Kandemir, and P. Raghavan. Intra-application cache partitioning. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, april 2010.

[98] Kyle J. Nesbit, James Laudon, and James E. Smith. Virtual private caches. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 57–68, New York, NY, USA, 2007. ACM.

[99] Christian Fensch Nick Barrow-Williams and Simon Moore. A communication characterisation of Splash-2 and Parsec. In *IISWC*, 2009.

[100] Derek B. Noonburg and John P. Shen. Theoretical modeling of superscalar processor performance. In *Proceedings of the 27th annual international symposium on Microarchitecture*, MICRO 27, pages 52–62, New York, NY, USA, 1994. ACM.

[101] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the 1993*

ACM SIGMOD international conference on Management of data, SIGMOD '93, pages 297–306, New York, NY, USA, 1993. ACM.

[102] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. MARSSx86: A full system simulator for x86 CPUs. In *Proceedings of Design Automation Conference*, DAC '11, 2011.

[103] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '03, pages 318–319, New York, NY, USA, 2003. ACM.

[104] M.D. Powell, A. Agarwal, T.N. Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, pages 54 – 65, dec. 2001.

[105] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Gated-vdd: a circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the 2000 international symposium on Low power electronics and design*, ISLPED '00, pages 90–95, New York, NY, USA, 2000. ACM.

[106] Seth H. Pugsley, Josef B. Spjut, David W. Nellans, and Rajeev Balasubramonian. Swel: hardware cache coherence protocols to map shared data onto shared caches. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 465–476, New York, NY, USA, 2010. ACM.

[107] M.K. Qureshi. Adaptive spill-receive for robust high-performance caching in cmps. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 45 –54, feb. 2009.

[108] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[109] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[110] Moinuddin K. Qureshi, M. Aater Suleman, and Yale N. Patt. Line distillation: Increasing cache capacity by filtering unused words in cache lines. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 250–259, Washington, DC, USA, 2007. IEEE Computer Society.

[111] Moinuddin K. Qureshi, David Thompson, and Yale N. Patt. The v-way cache: Demand based associativity via global replacement. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 544–555, Washington, DC, USA, 2005. IEEE Computer Society.

[112] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Architectural support for operating system-driven cmp cache management. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 2–12, New York, NY, USA, 2006. ACM.

[113] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. Reconfigurable caches and their application to media processing. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 214–224, New York, NY, USA, 2000. ACM.

[114] Rakesh Reddy and Peter Petrov. Cache partitioning for energy-efficient and interference-free embedded multitasking. *ACM Trans. Embed. Comput. Syst.*, 9(3):16:1–16:35, March 2010.

[115] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '90, pages 134–142, New York, NY, USA, 1990. ACM.

[116] D. Rolan, B.B. Fraguela, and R. Doallo. Adaptive set-granular cooperative caching. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1 –12, feb. 2012.

[117] Daniel Sanchez and Christos Kozyrakis. The zcache: Decoupling ways and associativity. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 187–198, Washington, DC, USA, 2010. IEEE Computer Society.

[118] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 57–68, New York, NY, USA, 2011. ACM.

[119] Andre Seznec. DASC cache. In *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, HPCA '95, pages 134–, Washington, DC, USA, 1995. IEEE Computer Society.

[120] Andre Seznec and Francois Bodin. Skewed-associative caches. In *PARLE '93 Parallel Architectures and Languages Europe*, volume 694 of *Lecture Notes in Computer Science*, pages 305–316. Springer Berlin / Heidelberg, 1993.

[121] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 336–349, New York, NY, USA, 2003. ACM.

[122] SPEC: Standard Performance Evaluation Corporation. SPEC CPU2000 and SPEC CPU2006. http://www.spec.org/.

[123] Harold S. Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *IEEE Trans. Comput.*, 41(9):1054–1068, September 1992.

[124] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *J. Supercomput.*, 28(1):7–26, April 2004.

[125] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pages 117–, Washington, DC, USA, 2002. IEEE Computer Society.

[126] Karthik T. Sundararajan, Timothy M. Jones, and Nigel Topham. A reconfigurable cache architecture for energy efficiency. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 9:1–9:2, New York, NY, USA, 2011. ACM.

[127] Karthik T. Sundararajan, Timothy M. Jones, and Nigel Topham. Energy-efficient cache partitioning for future cmps. In *Proceedings of the 21st ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, Minneapolis, MI, USA, 2012. ACM.

[128] K.T. Sundararajan, T.M. Jones, and N. Topham. Smart cache: A self adaptive cache architecture for energy efficiency. In *2011 International Conference on Embedded Computer Systems (SAMOS)*, pages 41 –50, july 2011.

[129] K.T. Sundararajan, V. Porpodas, T.M. Jones, N.P. Topham, and B. Franke. Cooperative partitioning: Energy-efficient cache partitioning for high-performance cmps. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1 –12, feb. 2012.

[130] David Tarjan, Michael Boyer, and Kevin Skadron. Federation: repurposing scalar cores for out-of-order instruction issue. In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 772–775, New York, NY, USA, 2008. ACM.

[131] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. Cacti 5.1. Technical Report HPL-2008-20. *HP Laboratories Palo Alto*, 2008.

[132] Keshavan Varadarajan, S. K. Nandy, Vishal Sharda, Amrutur Bharadwaj, Ravi Iyer, Srihari Makineni, and Donald Newell. Molecular caches: A caching structure for dynamic creation of application-specific heterogeneous cache regions. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 433–442, Washington, DC, USA, 2006. IEEE Computer Society.

[133] Maurice V. Wilkes. The memory gap and the future of high performance memories. *SIGARCH Comput. Archit. News*, 29(1):2–7, March 2001.

[134] Qiang Wu, Margaret Martonosi, Douglas W. Clark, V. J. Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 271–282, Washington, DC, USA, 2005. IEEE Computer Society.

[135] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[136] Yuejian Xie and Gabriel H. Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 174–183, New York, NY, USA, 2009. ACM.

[137] Yuejian Xie and Gabriel H. Loh. Scalable shared-cache management by containing thrashing workloads. In *Proceedings of the 5th international conference on High Performance Embedded Architectures and Compilers*, HiPEAC'10, pages 262–276, Berlin, Heidelberg, 2010. Springer-Verlag.

[138] Se-Hyun Yang, Babak Falsafi, Michael D. Powell, and T. N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pages 151–, Washington, DC, USA, 2002. IEEE Computer Society.

[139] Se-hyun Yang, Michael Powell, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Dynamically resizable instruction cache: An energy-efficient and high-performance deep-submicron instruction cache. *Purdue University*, 2000.

[140] Chenxi Zhang, Xiaodong Zhang, and Yong Yan. Two fast and high-associativity cache schemes. *IEEE Micro*, 17(5):40–49, September 1997.

[141] Chuanjun Zhang, Frank Vahid, and Roman Lysecky. A self-tuning cache architecture for embedded systems. In *Proceedings of the conference on Design, automation and test in Europe - Volume 1*, DATE '04, pages 10142–, Washington, DC, USA, 2004. IEEE Computer Society.

[142] Chuanjun Zhang, Frank Vahid, and Walid Najjar. Energy benefits of a configurable line size cache for embedded systems. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'03)*, ISVLSI '03, pages 87–, Washington, DC, USA, 2003. IEEE Computer Society.

[143] Chuanjun Zhang, Frank Vahid, and Walid Najjar. A highly configurable cache architecture for embedded systems. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 136–146, New York, NY, USA, 2003. ACM.

[144] Yan Zhang, Dharmesh Parikh, Karthik Sankaranarayanan, Kevin Skadron, and Mircea Stan. Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects. *Technical Report,CS-2003-05*, 2003.